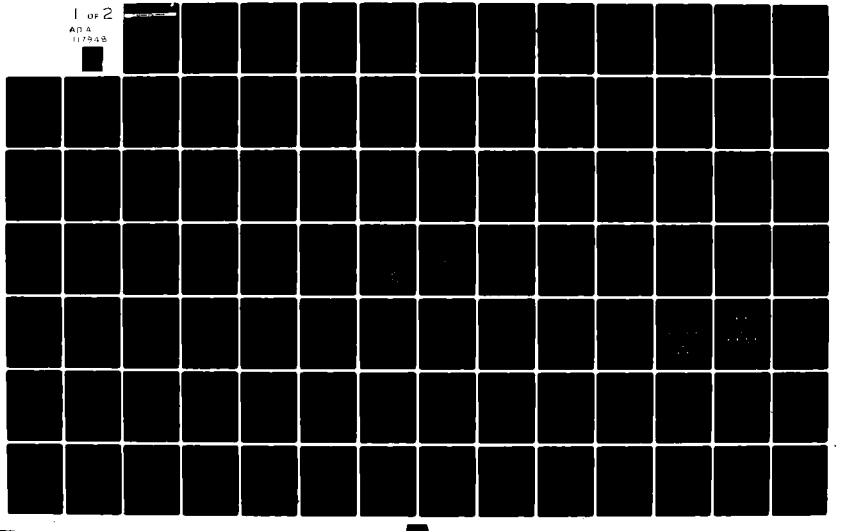
AD-A117 948 SYSTEMS CONTROL TECHNOLOGY INC PALO ALTO CA F/G 9/2
ENHANCEMENTS AND ALGORITHMS FOR AVIONIC INFORMATION PROCESSING --ETC(U)
JUN 82 K DOTY, A LEMOINE, P MCENTIRE N62269-81-C-0477
UNCLASSIFIED NADC-81105-50 NL



SCT SYSTEMS CONTROL TECHNOLOGY, INC.
1801 PAGE MILL RD. P.O. BOX 10180 PALO ALTO, CALIFORNIA 94303 (415) 494-2233

Report No. NADC-81105-50

ENHANCEMENTS AND ALGORITHMS FOR AVIONIC INFORMATION PROCESSING SYSTEM DESIGN METHODOLOGY

K. Doty
A. Lemoine
P. McEntire

SYSTEMS CONTROL TECHNOLOGY, INC.
1801 Page Mill Road
Palo Alto, CA 94304

16 June 1982

FINAL REPORT

Contract No. N62269-81-C-0477

Prepared for
NAVAL AIR DEVELOPMENT CENTER
Warminster, PA 18974

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: NADC-81105-50	2. GOVT ACCESSION NO.: AD-A117948
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): Enhancements and Algorithms for Avionic Information Processing System Design Methodology	5. TYPE OF REPORT & PERIOD COVERED: Final Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): K. Doty, A. Lemoine, P. McEntire	8. CONTRACT OR GRANT NUMBER(s): N62269-81-C-0477
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Systems Control Technology, Inc. 1801 Page Mill Road Palo Alto, CA 94304	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: F21-241-091 ZD121
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Air Development Center (5021) Warminster, PA 18974	12. REPORT DATE: 16 June 1982
	13. NUMBER OF PAGES: 123
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Distributed Data Processing, Software Allocation, Hardware Design, Dynamic Programming, Spatial Dynamic Programming, Stochastic Networks, Graph Theory

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report continues the study of both the software allocation and hardware configuration aspects of avionic information processing systems. The previously developed spatial dynamic programming algorithm is enhanced by incorporating task precedence constraints and hardware failures. Stochastic network methods are used to analyze allocations in the presence of random fluctuations. Graph theoretic methods are used to analyze hardware designs, and new designs are constructed with better values of important parameters, such as the graph diameter.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES (Continued)

SECTION I

INTRODUCTION

The objective of this contract is to begin development of a unified approach to the important problem of selecting an optimal hardware design for, and optimally allocating software tasks in, avionic information processing systems. This report summarizes the technical achievements accomplished during the contractual period.

The research conducted under this contract, and its predecessor [17], has focused on methodology for the combined software allocation/hardware design problem. The two aspects of the problem are inherently linked in that a hardware design cannot be evaluated properly until the software allocation is specified, and, conversely, that the "best" location for the software depends critically upon the hardware design.

The first step by Systems Control Technology (SCT) has been to look at the two aspects of the problem separately. In [17] the focus was on software allocation. There, spatial dynamic programming (SDP) was used to solve a static, deterministic software allocation problem. Under the current contract the SDP methodology has been extended to incorporate both dynamic and stochastic effects. In addition, an approach for evaluating software allocations in a stochastic environment has been developed which is quite flexible, yet simple and computationally tractable.

Investigation of the hardware design problem has also been initiated. A method based on graph theory has been used to evaluate candidate network architectures. Specific architectures have been identified which exhibit extremal values of important attributes such as maximum interprocessor distance. A more realistic hardware model involving buses over which several processors can communicate has also been studied.

Given these noteworthy and promising developments from our research, the next step is to merge the software allocation and hardware design

aspects of the problem to achieve the overall goal of the research program. In particular, for a specific set of software functions the optimal design and optimal allocation must be identified for given tradeoffs involving cost, performance, and availability. We anticipate that the results of this contract will be an integral part of the solution methodology for the combined problem.

This report is organized as follows. Section 2 addresses the software allocation problem. Each of the subsections here focuses on a separate aspect of the problem. Section 2.1 introduces the software allocation problem in some detail and also discusses the computational complexity of the SDP algorithm. Section 2.2 examines several methods for attacking the software allocation problem when the tasks are constrained by precedence relationships, as indeed they are in most avionic systems. Section 2.3 presents a methodology for evaluating allocations in the presence of stochastic fluctuations, using the methods of stochastic network theory. Section 2.4 discusses two extensions of the SDP methodology resulting in a more accurate model - one involving the allocation of files as well as tasks, the other allowing redundant assignment of tasks because of the possibility of hardware failures.

Section 3 concentrates on the hardware design problem. Section 3.1 presents an overview of the hardware design problem. In Section 3.2 a computer program called NETEV is described; this program has been used to evaluate candidate architectures. Section 3.3 discusses a particularly important problem in the design of networks which relates to minimizing the maximum inter-processor distance. Section 3.4 looks at the problem of network reliability, and the tradeoff between reliability and small distances. Finally, Section 3.5 examines the issue of how to design networks when buses connecting several processors are available.

The report also includes three appendices. Appendix A is a listing of the main allocation program using SDP; a sample set of input and output

is also provided. Appendix B is a listing of the NETEV program used to evaluate candidate computer network architectures. Appendix C provides a proof for a theorem presented in Section 3.4 involving processors with two ports.

Future Directions

The research described here is a first step to achieving the important goal of a complete software allocation/hardware design methodology to significantly improve the performance of avionic information processing systems. Several technical approaches for moving toward this unified methodology are currently envisioned. One of the simplest is a successive approximation method. For a given hardware arrangement, the optimal software allocation is identified, perhaps by a modification of the SDP algorithm in which variable hardware connections are allowed. Given the optimal software allocation, the optimal hardware connections are then found. The second step might be based on the work of Torng and Wilhelm [48], and may use results from Section 3 of this report. For the new hardware connection scheme, the optimal software allocation is found, and the process is repeated until the scheme converges.

Another approach is to blend the task-to-processor assignment graph with the bipartite hardware graph of Section 3.5. The result would be a tripartite graph, consisting of tasks, processors, and buses. Tasks would be assigned to processors, and processors assigned to buses. The solution method could again be based on SDP.

Whatever the solution method, solving the unified problem will produce significant payoffs. For example, the effect on an entire system of adding another processor can be investigated. Only if the advantages of improved performance and availability outweigh the increased cost should that processor be added. The capability for solving such allocation problems should indeed result in better avionic systems.

The methods developed should have a significant impact on the design of distributed computer systems, providing a quantitative foundation for decisions that vitally affect the performance and cost of the systems. The stringent constraints associated with avionic systems make such systems a fruitful application area for these methods.

SECTION II

THE SOFTWARE ALLOCATION PROBLEM

2.1 INTRODUCTION

We are interested in methods for the optimal allocation of a set of software tasks to a set of hardware elements consisting of processors, memories and communication links. This software allocation problem is difficult; it is a dynamic, stochastic, highly combinatorial problem with many different objectives.

SCT's approach has been to start with a simple model, and gradually increase the complexity to achieve greater realism. In the previous contract [17] a static, deterministic software allocation problem was considered in which a known set of tasks with known requirements and without precedence constraints are to be allocated to a set of fully connected processors with dedicated memories. The objective is to minimize the completion time of all the tasks. There are also constraints on the allocation; each task can only run on a subset of the processors, some tasks may have bounds on their finishing times, and memory constraints must be met.

The solution technique used for this problem has been spatial dynamic programming (SDP). Tasks and processors are considered as nodes of an allowable assignment graph. Nodes are processed one at a time, to minimize the completion time of the processor nodes considered thus far.

The code for the SDP algorithm appears as Appendix A of this report. It is written in PASCAL for a DEC VAX 11/780 and consists of 7 modules, which are then linked in order to execute the program. The heart of the code is procedure FIND COST, which computes the maximum processor finishing time. After the code listing a sample set of input and output appears for an example involving 8 processors and 20 tasks.

The amount of time required to solve a software allocation problem depends critically upon the complexity of the task-to-processor assignment graph. Given the assignment graph, there is the subproblem of finding the optimal order to process the nodes in the SDP method. Appendix B of [17] discusses this subproblem in the case where each of n tasks can run on each of m processors. The problem complexity is then approximately $2^{mn/2}$.

This is an upper bound on the complexity, because often there will be sparsity in the assignment graph. Using the notation of [39], the complexity is approximately equal to

$$\sum_{k=1}^{m+n} 2^{|E_k| + |\hat{I}_k|} \tag{1}$$

where E_k and $\hat{I}_k$ are sets of external and internal variables involved with the kth node. If the values of $|E_k| + |\hat{I}_k|$ are approximately equal, as they will be in the optimal order, then expression (1) is close to

$$(m+n)\ 2^{\max(|E_k| + |\hat{I}_k|)} \tag{2}$$

Now it is still difficult to tell what $|E_k| + |\hat{I}_k|$ is. About all that is known is that $|E_k| + |\hat{I}_k| \geq d_k$, where d_k is the degree of the kth node. Suppose each task can be allocated to c processors. Then some processor must be able to handle cn/m tasks. As a result, $\max(|E_k| + |\hat{I}_k|) \geq cn/m$. Thus expression (2) becomes approximately

$$(m + n)2^{cn/m} \tag{3}$$

assuming that $n > m$. Expression (3) shows qualitatively what has been experienced with the SDP algorithm in practice. If the number of tasks and processors both increase without changing their ratio, then the amount of work increases linearly. However, if the number of tasks alone increases, or the graph becomes denser (c increases), then the amount of work increases exponentially.

The software allocation problem for avionic systems is dynamic and stochastic. Certain tasks must execute before others. Several methods for extending the SDP algorithm to cover the allocation problem with precedence relations are discussed in Section 2.2. In addition, there may be uncertainty about exactly which tasks must execute, or how long they will take. Section 2.3 examines a method for analyzing the behavior of processing demands in a stochastic environment, using the methods of stochastic network theory. These methods of analysis can be used to test the effectiveness of a software allocation algorithm.

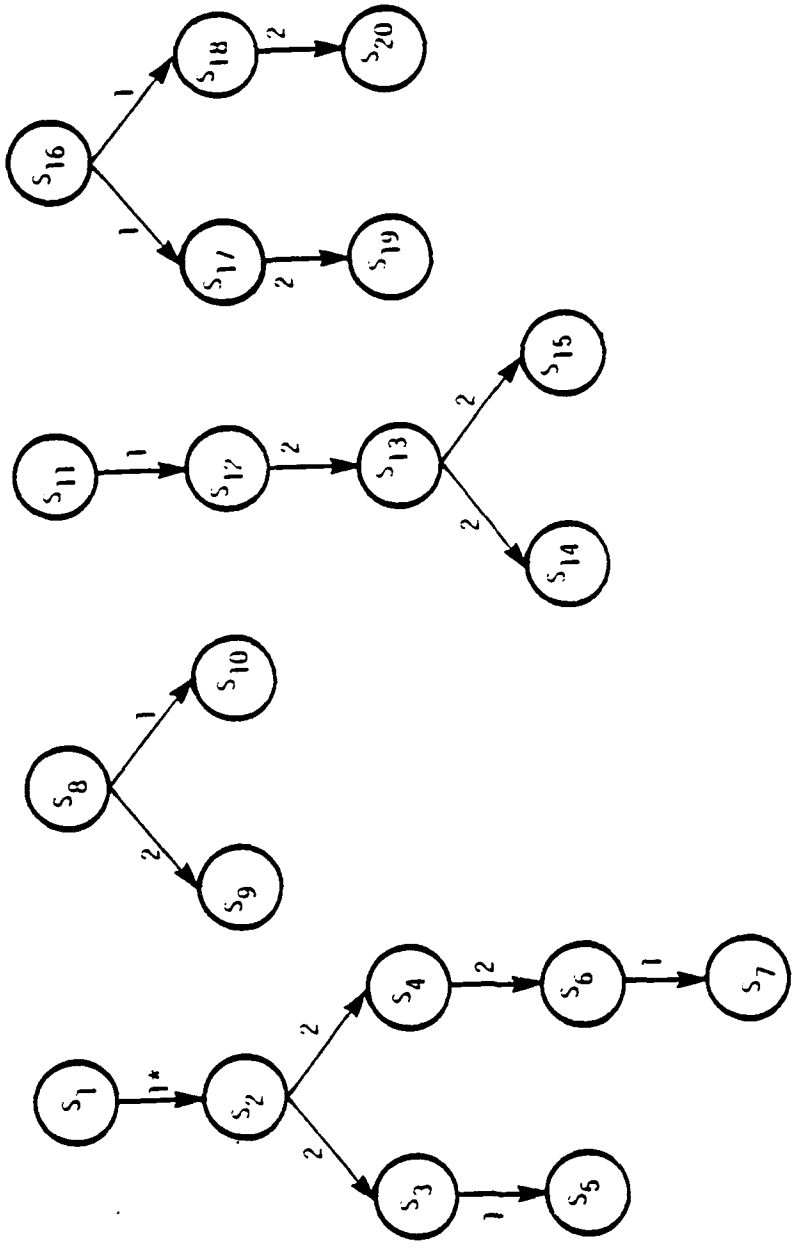
The last section (2.4) discusses two modifications of the basic SDP algorithm to correspond to a more realistic software/hardware model. The first modification concerns the allocation of files as well as tasks. The other modification allows the possibility of processor failures. Tasks can be allocated to more than one processor, with the more important tasks being the ones most likely to have redundant assignments.

2.2 SOFTWARE ALLOCATION WITH PRECEDENCE CONSTRAINTS

The tasks for execution in real avionic software systems usually have precedence constraints. That is, certain tasks must finish executing before others are allowed to start. The principal reason for such constraints is that data from earlier tasks is needed for those coming later.

A solution for the allocation problem with precedence constraints requires not only an allocation of the tasks to the processors, but also a scheduling of the tasks. The time that each task begins executing must be specified and when a task starts, all tasks which come before it (as dictated by the precedence constraints) must be finished.

Figure 2.1 shows an example of precedence relationships among a set of 20 tasks. (This example is taken from [17]). A feasible schedule for these tasks on a complete network of six processors is shown in Figure 2.2. Notice that some processors have "dead time," in which no task is being executed.



*Intermodule communication (x1000 bytes)

Figure 2.1. Software Trees for Example

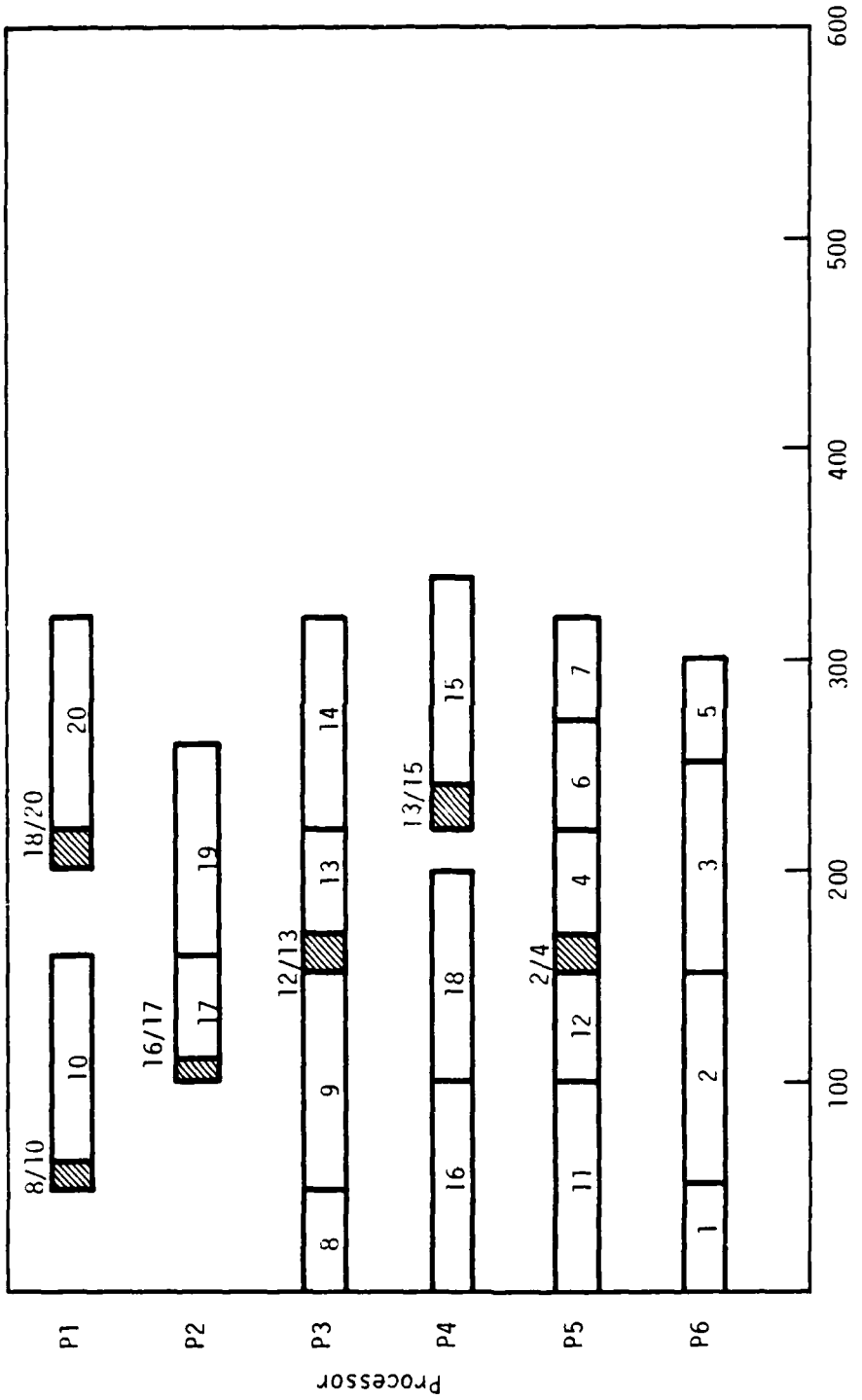


Figure 2.2. Schedule for Example

The precedence problem is fundamentally computationally intensive. Thus, any solution method will either grossly simplify the problem, or require an excessive amount of computation, or use heuristics to arrive at a solution which is most likely sub-optimal. As a result, it seems advisable to discuss several different approaches to this problem, each of which has its own advantages and drawbacks. However, we will recommend one method (spatial dynamic programming with the "method of windows") which we believe is the best approach.

The classical methods for solving such problems can be found in the area of operations research known as scheduling theory. Generally speaking, a set of tasks is to be scheduled on one or more machines, in order to optimize some performance criterion. Unfortunately, all but the simplest formulations fall into a class of problems known as NP-complete, which are notorious for their computational intractability. For example, Hu [32] showed that if there are m identical processors, n tasks which each require one time unit, and if the precedence constraints form a tree, then there is a simple algorithm to minimize the completion time of all the tasks. If the tasks are not identical in time requirements, or if the precedence constraints do not form a tree, then the problem is NP-complete [38]. Lenstra and Rinooy Kan [38] discuss several solvable precedence problems and many more unsolvable ones (at least in a reasonable amount of time).

Only a few of the papers on software allocation discuss precedence constraints. One such paper [8] is the most recent in a series of articles by Kokhari. In [8] the precedence relationships must again be in the form of a tree. Each task is allocated to a processor and to a time phase. If task i must preceed task j and task i is allocated to a time phase k, then task j must be allocated to the same phase or later. The solution method, based on dynamic programming, finds the optimal solution subject to the predetermined phases. In this case "optimal" has a different meaning - the sum of execution and communication times. As a

result, the solutions (such as the example in [8]) are likely to have a significant amount of dead time, which may not be acceptable. Another drawback to this method is the difficulty of incorporating memory constraints. Thus while the method may be interesting, it is unlikely to solve our problem.

Dr. Ma and others at TRW used the branch-and-bound method of integer programming to solve an allocation problem in [40]. The method involved a form of precedence constraint in that several "threads" of tasks had to satisfy "port-to-port" time constraints. A "thread" consists of a set of tasks which must execute consecutively. However, the method for determining whether these time constraints are met is not explained. In addition, the optimization criterion of minimizing interprocessor communication may result in unbalanced processor loads.

Instead of trying to find an optimal solution to a simple formulation, a suboptimal solution to a complex formulation may be found via heuristics. One heuristic method for the allocation problem with precedence constraints is described in the final report of this contract's predecessor [17]. It is based on the longest-path method used by Kaufman [34]. The modifications described in our report incorporate inter-task communication and task memory requirements. Several examples are given there showing how the method is used. The primary drawback to this method is that because it is heuristic it is unlikely to produce optimal solutions. In many cases the solutions will be near-optimal, but sometimes they may be far from optimal. However, for problems which are large enough (say, 15-20 tasks per processor) that more complex methods (such as the "method of windows" described below) are computationally prohibitive, a good heuristic method such as this is essentially the only way to produce an acceptable allocation.

The method we recommend for solving the allocation problem with precedence constraints is spatial dynamic programming, with certain modifications. The primary reason for this is that the SDP method is very general. Several different performance criteria and many types of constraints can be incorporated.

The modifications are called the "method of windows." For each task we set up a "window" of time in which it may execute. The window is larger than the task's execution time. If task A must precede task B, A's window should come before B's. Any allocation which satisfies the window constraints automatically satisfies the precedence constraints.

Figure 2.3 shows an example of a precedence tree. Assume that this is one of several in a particular allocation problem. The execution times are given in the second column of Table 2.1. Next the windows need to be established. The last column of Table 2.1 gives one set of windows which forces the tasks to satisfy the precedence constraints.

Finding an appropriate size for the windows must be done on a heuristic basis. Factors which should be taken into account include the number of processors and the amount of parallelism in the software structure. If the windows are too small there will be no feasible solution, and the windows should be enlarged. If the windows are too big there will be a significant amount of dead time in the schedule, making the completion time larger than it should be. This can be remedied by either compressing the schedule to remove some dead time or by shrinking the window sizes.

Once the windows are established, the SDP algorithm proceeds as in the static case with no precedence relationships. However, now there are additional constraints which must be checked at each processor node. For each set of tasks which may be allocated to the processor, it must be determined whether there is a feasible solution which meets the window constraints. If there are k tasks in the candidate allocation, there are $k!$ possible execution orders. Conceivably each of these may have to be examined to see if there is one which meets the constraints.

As an example, consider the situation shown in Figure 2.4. Only one order of execution, A then C then B, will satisfy the window constraints. However, if B required more than 15 time units there would be no feasible schedule.

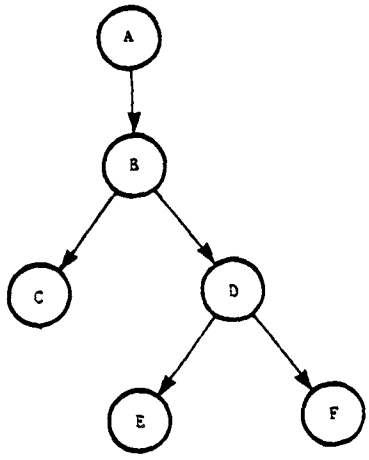


Figure 2.3. Example of Precedence Relationships

Task	Execution Time	Window
A	10	0-15
B	20	15-45
C	10	45-65
D	15	45-70
E	25	70-110
F	5	70-90

Table 2.1
Execution Times and Corresponding Windows

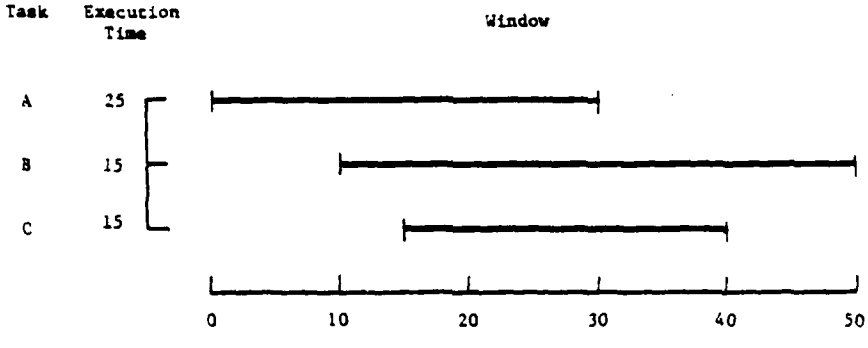


Figure 2.4. Example of Task Windows

If k is not more than, say 10, it is feasible to consider all possible orders. McMahon and Florian [41] give an efficient method of enumerating all possibilities. For larger values of k, heuristic methods may have to be used. Baker and Su [4] showed that some heuristics can often determine whether a feasible schedule exists. If the heuristics do not come up with a feasible solution, the windows may have to be enlarged.

As an example of how the method of windows works, consider the two software functions shown in Figure 2.5. (Admittedly this is a small example, but it will illustrate the method). The data for the tasks is shown in Table 2.2. The windows were chosen to be about 50% larger than the execution times, and to preserve the precedence constraints. The hardware will be three fully connected processors, each with a dedicated memory with capacity 40. Figure 2.6 shows the network of possible assignments.

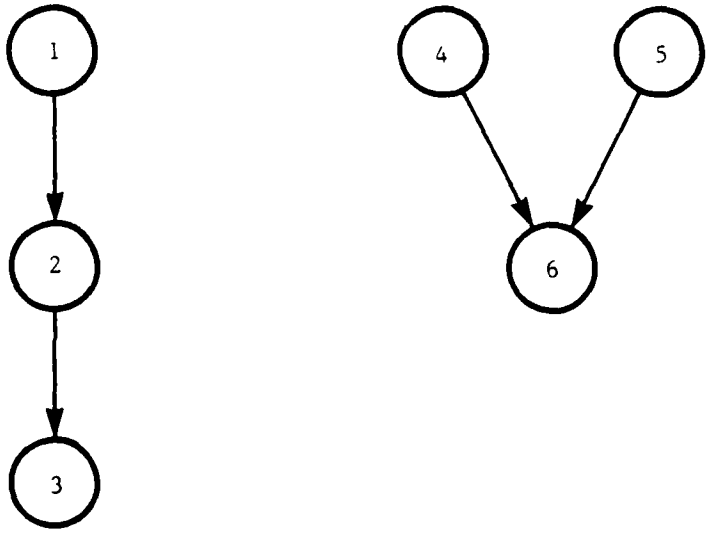


Figure 2.5. Two Software Functions with Precedence Constraints

TASKS	EXECUTION TIME	WINDOW	MEMORY REQUIREMENT
1	10	0-15	20
2	5	15-25	10
3	15	25-50	20
4	10	0-30	25
5	20	0-30	10
6	15	30-50	10

Table 2.2
Data for Example

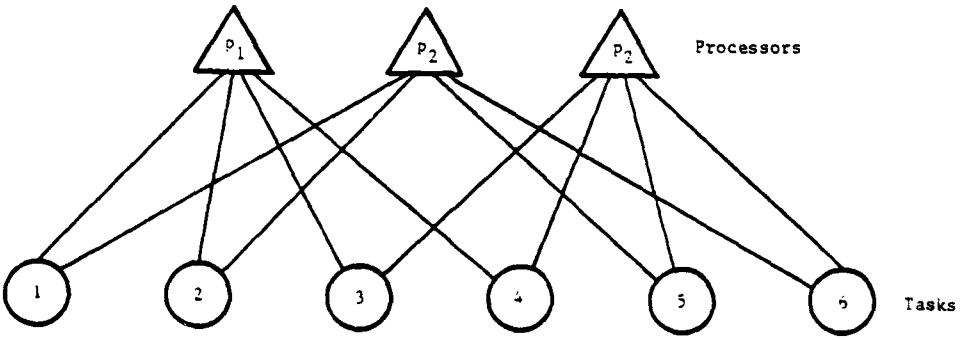


Figure 2.6. Network of Possible Assignments

One point to notice is that the window constraints are binding. For example, there is sufficient memory for tasks A, B, and E all to be located at processor 2. However, not all of the task window constraints can be met.

Using SDP, an optimal allocation given these windows is found to be assigning task 4 to processor 1, tasks 1, 2, and 6 to processor 2, and tasks 3 and 5 to processor 3. The resulting schedule is shown in Figure 2.7. As expected, there is a significant amount of dead time. The schedule can be compressed, being careful to maintain the precedence constraints, as shown in Figure 2.8.

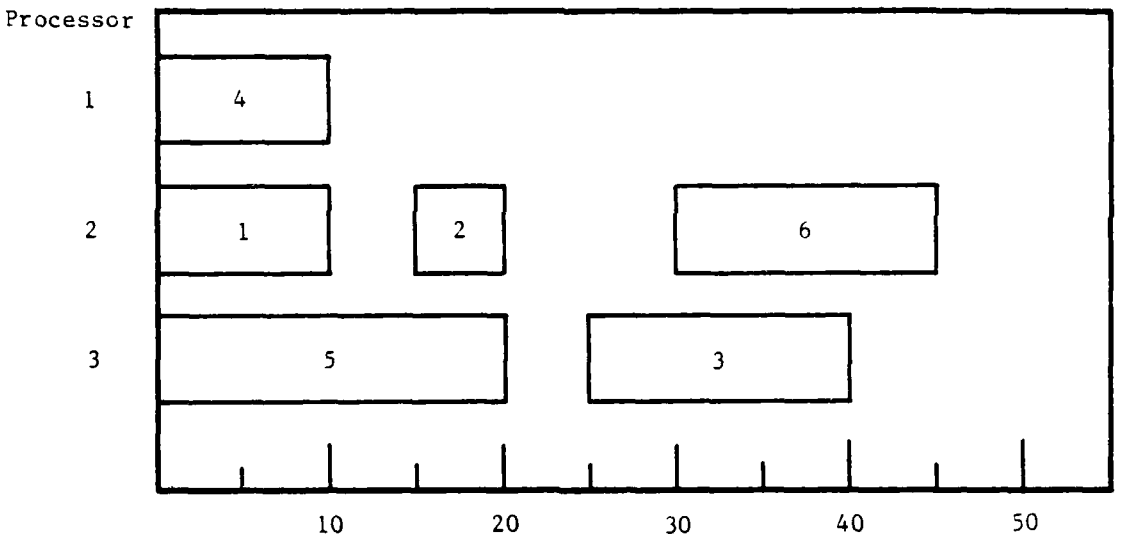


Figure 2.7. Schedule Produced by SDP Algorithm and "windows"

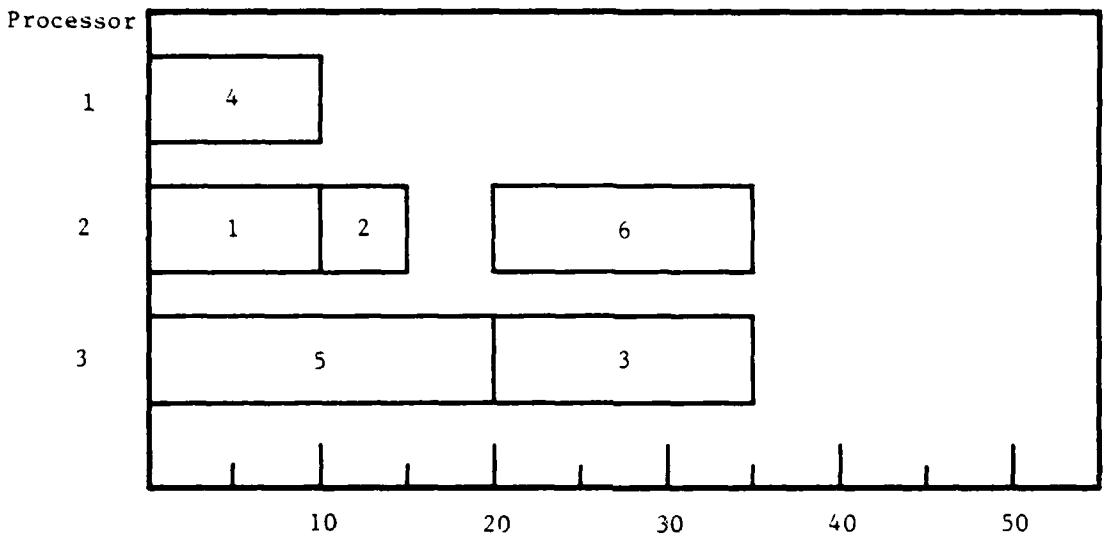


Figure 2.8. Compressed Schedule

We have examined several methods for attacking the software allocation problem when there are precedence constraints. Another important feature in avionic systems is that the tasks needed on a mission and their execution times are uncertain. The next section discusses a method for evaluating allocations in a stochastic environment.

2.3 STOCHASTIC VARIABILITY AND SOFTWARE ALLOCATION

We are concerned in this section with a network of processors and associated software functions for which the input of jobs (requests for execution of software functions) is uncontrolled, movements (routes) of jobs through the system and class changes of jobs are random, resources required to process jobs vary randomly, time horizons of shorter duration than necessary to achieve "steady-state" are of interest, and where it is important to respond to demands with minimal competition for available

processing resources. We want to account for these various factors through an appropriate network model and provide useful and computationally tractable results for performance prediction. The model structure we consider is reminiscent of Baskett et al [7] but the approach taken in analyzing the model is prompted by Harrison and Lemoine [29].

In this model, a "job" corresponds to a sequence of tasks. As the job moves through the network of processors, different tasks are being executed. Changing the "class" of a job corresponds to moving from one task to another within a job. Admittedly this is different terminology than has appeared previously; it is used to correspond to the terminology in the stochastic network theory literature.

This section is organized as follows. The basic features of the model are described in 2.3.1. Following this, we consider in 2.3.2 the movement of a typical job through the network with particular emphasis on the resources required, global as well as nodal, to process the job until it leaves the system. We then consider in 2.3.3 the evolution of the network over time when the pattern of demand input is a Poisson process and the processing resources available at each node in the network are effectively unlimited (e.g., there are sufficient servers at each node so that objects never wait in queue). An explicit representation is given for the time-dependent distribution of network state (defined here as the numbers of jobs of various types at each node in the system). When the Poisson input process is homogeneous, this time-dependent distribution converges to a limit, and the "distance" between the time-dependent and limiting distributions is estimated. Some implications of the results for design and performance analysis are indicated. Indeed, the basic notion behind the approach of 2.3.3 is to first determine a (time-dependent) distribution on the level of resource usage in the system assuming ample resources are available, and to then use this distribution for identifying candidate resource capacity design parameters and assessing "adequacy" via confidence intervals obtained from the distribution.

2.3.1 Model Description

Consider a network in which jobs in different classes are moving through the system, and also possibly changing class, in a possibly random manner, and requiring a possibly random amount of resource (program, processing, memory, time, etc.) at each node visited. There are N nodes (processing centers, clusters, etc.) indexed by i and M job classes indexed by m. Inputs for processing at any node may originate directly from outside the system or by internal transfer within the network. On any visit to node i by a job in class m the job is completed (routed to an artificial sink in the network) after processing at i with probability q_{mi}, independently of previous visits, classes and other jobs present in the system; and, in this event, the amount of resource required at node i to process the job is distributed as a random variable S_{mi}. Further, on any visit to node i by a job in class m the job remains in the system after processing at node i and transfers to node j as a job in class r with probability p_{ij}^{mr}, independently of prior visits, classes and other jobs (if i=j then the job is sent back to node i), where

$$q_{mi} = 1 - \sum_{j=1}^{N} \sum_{r=1}^{M} p_{ij}^{mr}$$

for each m and i; and, in this event, the amount of resource required at node i to process the job is distributed as a random variable R_{ij}^{mr}. Processing requirements for all visits by all jobs in all classes to any node are independent, and all jobs eventually exit from the system. The model dynamics can be summarized by two examples. The top of Figure 2.9 shows two software functions A and B with separate external input. In function A, there is a stochastic transition after task 1: task 2 is executed with probability p and task 3 is executed with probability 1-p. In function B, after task 7 finishes the function is again executed with probability 1-q. This might correspond to a tracking function, where the tracking stops if the object being tracked leaves the radar screen.

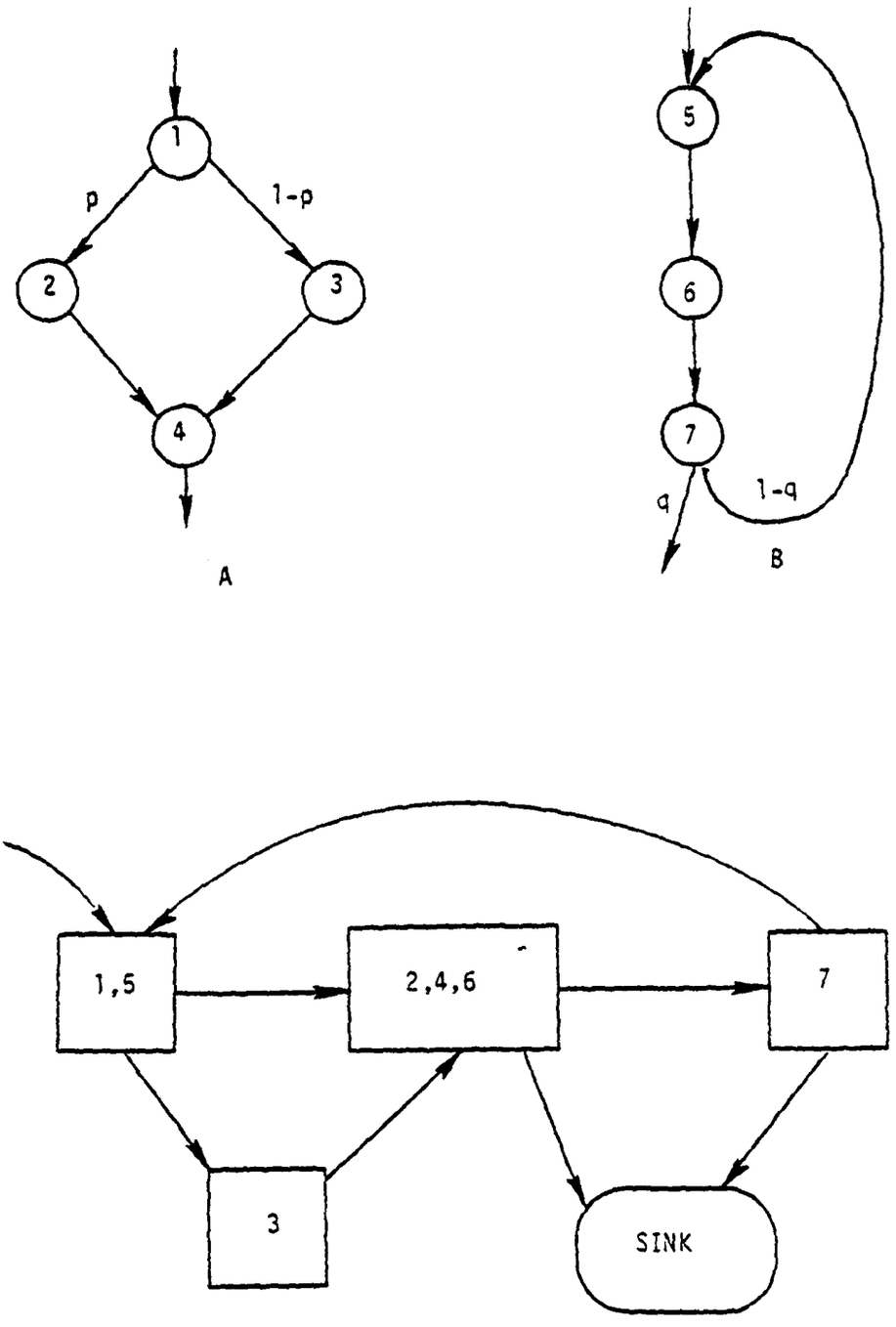


Figure 2.9. Network Examples

The bottom of Figure 2.9 shows a possible allocation of these tasks to a network of four processors. The transitions which are actually made depend on the particular task being executed at a processor. For example, from the processor where tasks 1 and 5 are allocated execution proceeds to the right if either task 2 or task 6 follows, while it goes to the processor below if task 3 follows. All execution streams eventually enter the sink with probability 1.

The preceding formulation can easily accommodate multiple resource types by specifying, for each type of resource, appropriate probability distributions for R_{ij}^{mr} and S_{mi} for $m,r=1,\ldots,M$ and $i,j=1,\ldots,N$. In what follows we do not distinguish between multiple resource types in order to keep the notation from being excessively complicated, but the results presented on resource requirements are valid for each resource type.

2.3.2 Routing and Resource Requirements

Consider now the route and class of a generic job moving through the network. Let b_{mi} be the probability that a job enters the system through node i in class m. For $i,j=1,\ldots N$ let $P_{ij} = [p_{ij}^{mr}]$ where $m,r=1,\ldots,M$ so that P_{ij} is a M by M matrix, and then put $P = [P_{ij}]$ so that P is a N by N block matrix. Let I be the NM by NM identity matrix. The routing-and-class history of a typical job moving through the system corresponds to the evolution of a finite Markov chain with a single absorbing state (namely, the network sink) and non-absorbing states $\{(m,i): m=1,\ldots,M \text{ and } i=1,\ldots,N\}$, with transitions among non-absorbing states governed by P. Since all jobs eventually exit the system (i.e., the chain is absorbing) the matrix $I - P$ is invertible, and

$$[I - P]^{-1} = \sum_{x=0}^{\infty} P^x .$$

If we put $(I - P)^{-1} = [A_{ij}]$ and $A_{ij} = [a_{ij}^{mr}]$ then a_{ij}^{mr} is the expected number of requests for processing at node j by a job in class r which

originally enters the network through node i in class m. Let ρ_{ij}^{mr} be the probability a job entering via node i in class m reaches node j in class r at some later step. Then $\rho_{ii}^{mm} = 1 - (a_{ii}^{mm})^{-1}$ and $\rho_{ij}^{mr} = a_{ij}^{mr}/a_{jj}^{rr}$ otherwise. Moreover, if q_{ij}^{mr} is the probability that a job entering the network via node i in class m exits the system from node j in class r then $q_{jj}^{rr} = q_{rj}/(1 - \rho_{jj}^{rr})$ and $q_{ij}^{mr} = \rho_{ij}^{mr} q_{jj}^{rr}$ otherwise. Finally, if q_{mi}^{*} is the unconditional probability that a job exits the system from node i in class m then

$$q_{mi}^{*} = q_{mi} \sum_{j=1}^{N} \sum_{r=1}^{M} b_{rj} a_{ji}^{rm} .$$

The above expressions for the probabilities ρ_{ij}^{mr}, q_{ij}^{mr} and q_{mi}^{*} follow immediately from well known results for finite absorbing Markov chains; cf. Kemeny and Snell [35]. Observe that the crucial factor in determining these various routing and class parameters is computation of the matrix $(I - P)^{-1}$. In many applications of interest this matrix $I - P$ will be relatively sparse, indeed even upper triangular.

Turning to global network resource requirements, let L_{mi} be the total level of resources necessary to process a job until it reaches the network sink given that the job enters the system at node i in class m. For $\theta \geq 0$ let $f_{mi}(\theta)$, $g_{ij}^{mr}(\theta)$, and $h_{mi}(\theta)$ be the Laplace transforms for the distributions of S_{mi}, R_{ij}^{mr}, and L_{mi}, respectively. By virtue of the independence assumptions regarding routing and class and resource requirements on visits to nodes, the transforms $\{h_{mi}(\theta)$: $m = 1,\ldots,M$ and $i = 1,\ldots,N\}$ satisfy the network flow equations

$$h_{mi}(\theta) = q_{mi} f_{mi}(\theta) + \sum_{j=1}^{N} \sum_{r=1}^{M} p_{ij}^{mr} g_{ij}^{mr}(\theta) h_{rj}(\theta) \qquad (1)$$

for $m=1,\ldots,M$ and $i=1,\ldots,N$. This system of equations has a unique solution as follows: Let $f(\theta)$ and $h(\theta)$ be NM column vectors consisting of N strings of length M where entry m of string i in $f(\theta)$ is $q_{mi} f_{mi}(\theta)$ and entry r of string j in $h(\theta)$ is $h_{rj}(\theta)$.

Let $D(\theta)$ be a N by N block matrix where the $(ij)^{th}$ block, say $D_{ij}(\theta)$, is the M by M matrix $[p_{ij}^{mr}\, g_{ij}^{mr}(\theta)]$. Then (1) is equivalent to

$$h(\theta) = f(\theta) + D(\theta)\, h(\theta) \tag{2}$$

or

$$[I - D(\theta)]\, h(\theta) = f(\theta).$$

Since $0 \leq D(\theta) \leq P$ component-wise it follows that the matrix $[I - D(\theta)]$ is invertible for any $\theta \geq 0$; indeed,

$$[I - D(\theta)]^{-1} = \sum_{x=0}^{\infty} [D(\theta)]^x .$$

Thus

$$h(\theta) = [I - D(\theta)]^{-1} f(\theta). \tag{3}$$

Computation of the transforms $\{h_{mi}(\theta)\}$ from (3) is no simple matter, but we will provide an explicit representation for these transforms using a slightly different approach.

On the other hand, repeated differentiation of the network flow equations with respect to θ, and then setting $\theta = 0$, provides a recursive procedure for computing the moments (first, second, third, and so on) of the cumulative resource variables $\{L_{mi}\}$. For $n = 1,2,\ldots$ let $h_{mi}(n) = E\{(L_{mi})^n\}$, $g_{ij}^{mr}(n) = E\{(R_{ij}^{mr})^n\}$ and $f_{mi}(n) = E\{(S_{mi})^n\}$. We assume henceforth that R_{ij}^{mr} and S_{mi} have finite moments of all orders for $m,r=1,\ldots,M$ and $i,j=1,\ldots,N$; the distributions customarily used in the modeling of service systems have this property, and so the assumption is not really restrictive. We will now observe that the variables $\{L_{mi}\}$ also have finite moments of all orders and provide a recursive procedure for computing these moments.

Differentiating both sides of (1) n times with respect to θ and using Leibnitz's Rule leads to

$$h^{(n)}(\theta) = f^{(n)}(\theta) + \sum_{x=0}^{n} \binom{n}{x} D^{(x)}(\theta) h^{(n-x)}(\theta),$$

or

$$h^{(n)}(\theta) = [I - D(\theta)P]^{-1} \left(f^{(n)}(\theta) + \sum_{x=1}^{n} \binom{n}{x} D^{(x)}(\theta) h^{(n-x)}(\theta) \right) \qquad (4)$$

where the superscripts correspond to component-wise differentiation in (2) and

$$\binom{n}{x} = \frac{n!}{x!(n-x)!} .$$

Suppose that each of the variables $\{(L_{mi})^{n-1}\}$ has finite mean (this certainly holds for $n = 1$). Then each component of $h^{(n-x)}(\theta)$ has a finite limit as $\theta \to 0$ for $x=1,\ldots,n$. Since $[I - D(\theta)]^{-1} \to [I - P]^{-1}$ as $\theta \to 0$, it now follows from (4) that each component of $h^{(n)}(\theta)$ has a finite limit as $\theta \to 0$, and so each of the variables $\{(L_{mi})^{n}\}$ has finite mean. Thus, we conclude by mathematical induction that each of the variables $\{L_{mi}\}$ has finite moments of all orders. Moreover, using (4) we can give a compact recursive formula for computing these moments. For $x=1,2,\ldots$ let δ_x, h_x and h_o be NM column vectors where the entries corresponding to (m,i) are $q_{mi}f_{mi}$, h_{mi}, and 1, respectively; also, let D_x be a NM by NM matrix where the entry corresponding to (m,i), (r,j) is $p_{ij}^{mr} g_{ij}^{mr}(x)$. Then

$$h_n = [I - P]^{-1} \left[\delta_n + \sum_{x=1}^{n} \binom{n}{x} D_x h_{n-x} \right] \qquad (5)$$

for $n=1,2,\ldots$. The notable feature of the recursion formula (5) is that it requires only the given network parameters (i.e., the matrix P and the moments of the variables $\{R_{ij}^{mr}, S_{mi}\}$) and computation of the matrix $[I - P]^{-1}$.

With regard to nodal (or local) resource requirements, let L_{ij}^{mr} be the total level of resources required at node j to process a job in class r which enters the network at node i in class m. Put another way, L_{ij}^{mr}

is the total loading induced at node j by an object in class r entering at node i in class m. Observe that

$$L_{mi} = \sum_{j=1}^{N} \sum_{r=1}^{M} L_{ij}^{mr} .$$

Let $\phi_{ij}^{mr}(\theta)$ be the Laplace transform for the distribution of L_{ij}^{mr}. For an arbitrary node k and job class s let $q_{sk} + u_{sk} = 1 - \rho_{kk}^{ss}$ and

$$\phi_{sk}(\theta) = \frac{q_{sk} f_{sk}(\theta) + u_{sk} g_{sk}^{sk}(\theta)}{1 - \rho_{kk}^{ss} g_{sk}^{sk}(\theta)} .$$

The quantity u_{sk} is the probability that a job at node k in class s departs the network either without returning to node k or from node k but in a different class. We then have

$$\phi_{ij}^{mr}(\theta) = \begin{cases} \phi_{mi}(\theta) & \text{if } i=j \text{ and } m=r, \text{ and} \\ 1 - \rho_{ij}^{mr} + \rho_{ij}^{mr} \phi_{rj}(\theta) & \text{otherwise .} \end{cases}$$

Moreover, in reference to the comment following (3) above,

$$h_{mi}(\theta) = \prod_{j=1}^{N} \prod_{m=1}^{M} \phi_{ij}^{mr}(\theta).$$

The transforms $\{\phi_{ij}^{mr}(\theta)\}$ as given above make possible computation of the moments of the variables $\{L_{ij}^{mr}\}$. For example:

$$E\{L_{ii}^{mm}\} = \frac{q_{mi} f_{mi}(1) + (1-q_{mi}) g_{mi}^{mi}(1)}{1 - \rho_{ii}^{mm}} ,$$

$$E\{(L_{ii}^{mm})^2\} = \frac{q_{mi} f_{mi}(2) + (1-q_{mi}) g_{mi}^{mi}(2) + 2\, \rho_{ii}^{mm} g_{mi}(1)\, E\{L_{ii}^{mm}\}}{1 - \rho_{ii}^{mm}} ,$$

$$E\{L_{ij}^{mr}\} = \rho_{ij}^{mr} E\{L_{ii}^{mm}\} ,$$

$$E\{(L_{ij}^{mr})^2\} = \rho_{ij}^{mr} E\{(L_{ii}^{mm})^2\} .$$

Finally, let L_{mi}^{*} be the total resources required at node i in class m by a typical job entering the network. Then the distribution of L_{mi}^{*} has Laplace transform

$$\sum_{j=1}^{N} \sum_{r=1}^{M} b_{rj} \phi_{ji}^{rm}(\theta) .$$

2.3.3 Poisson Input

We consider now the evolution of the network over time when jobs enter the system according to a Poisson process $A = \{A(t), t \geq 0\}$ with intensity function $\{\lambda(t), t \geq 0\}$ and when the processing resources available to each node in the network are effectively unlimited (e.g., in queueing parlance, there are sufficient servers available at each node so that jobs never wait in queue). The variables R_{ij}^{mr} and S_{mi} are then interpreted as processing times at node i. The discussion we give follows Section 3 in Harrison and Lemoine [29].

The assumption of Poisson input as stated above means that the numbers of jobs arriving in non-overlapping time intervals are independent random variables and that for $n = 0,1,2,\ldots$

$$P\{A(t) = n\} = e^{-\Lambda(t)} \frac{[\Lambda(t)]^n}{n!} , \qquad t \geq 0,$$

where

$$\Lambda(t) = \int_0^t \lambda(y)dy .$$

In particular, if $A_{mi}(t)$ is the number of jobs entering the network through node i in class m up to time t and $A_{mi} = \{A_{mi}(t),\ t \geq 0\}$, then $\{A_{mi}: m=1,\ldots,M \text{ and } i=1,\ldots,N\}$ are independent Poisson processes and A_{mi} has intensity function $\{b_{mi}\lambda(t),\ t \geq 0\}$.

The histories (routes and classes) of jobs moving through the system are independent and distributed as a process $Y = \{Y(t),\ 0 \leq t \leq L\}$ where $Y(t) \in \{(m,i): m=1,\ldots,M \text{ and } i=1,\ldots,N\}$ with the variable L interpreted as the total length of time a generic job is in the network. In particular, $Y(0) = (m,i)$ with probability b_{mi}, in which case L has the same distribution as the variable L_{mi}. Now define

$$\gamma_{rj}(t) \equiv P\{L > t,\ Y(t) = (r,j)\}$$

$$= \sum_{i=1}^{N} \sum_{m=1}^{M} b_{mi} P\{L_{mi} > t,\ Y(t) = (r,j) \mid Y(0) = (m,i)\},$$

$$\xi_{rj}(t) \equiv \int_0^t \lambda(y)\gamma_{rj}(t-y)dy,$$

and

$$\xi(t) \equiv \sum_{j=1}^{N} \sum_{r=1}^{M} \xi_{rj}(t) = \int_0^t \lambda(y)P\{L > t-y\}dy .$$

Assuming the network is empty at time 0, the quantity $\xi_{rj}(t)$ is the expected number of jobs occupying node j in class r at time t and $\xi(t)$ is the expected number of jobs in the system.

Now let $C_{rj}(t)$ be the number of jobs occupying node j in class r at time t and let

$$\mathscr{C}(t) = \{C_{rj}(t): r=1,\ldots,M \text{ and } j=1,\ldots,N\}$$

be the "state" of the network at time t. Also, let $C = \{c_{rj}\}$ be a generic state of the system. Then the following remarkable result holds (cf. [29]):

$$P\{\mathscr{C}(t) = C\} = e^{-\xi(t)} \prod_{j=1}^{N} \prod_{r=1}^{M} [\xi_{rj}(t)]^{c_{rj}}/c_{rj}! \quad . \tag{6}$$

That is, the number of jobs occupying node j in class r at time t has a Poisson distribution with mean $\xi_{rj}(t)$, and the numbers occupying the various nodes and classes are independent random variables.

Implementation of (6) requires computation of the mean values $\{\xi_{rj}(t)\}$ which is difficult for non-homogeneous input. Suppose, however, that A is *a homogeneous Poisson process* with $\lambda(t) \equiv \lambda$ for $t \geq 0$. Let

$$\xi_{rj} = \lambda \int_0^\infty \gamma_{rj}(y)dy = \lambda \sum_{i=1}^{N} \sum_{m=1}^{M} b_{mi} E\{L_{ij}^{mr}\}$$

and

$$\xi = \sum_{j=1}^{N} \sum_{r=1}^{M} \xi_{rj} \quad .$$

In the homogeneous case we have $\xi_{rj}(t) \to \xi_{rj}$ as $t \to \infty$. It then follows from (6) that

$$\lim_{t\to\infty} P\{\mathscr{C}(t) = C\} = e^{-\xi} \prod_{j=1}^{N} \prod_{r=1}^{M} (\xi_{rj})^{c_{rj}}/c_{rj}! \quad . \tag{7}$$

This limiting or asymptotic distribution depends only on the arrival rate λ and the <u>expected</u> amount of time ξ_{rj} that a job spends at node j in class r while in the network, and <u>not</u> upon the forms of the distributions for processing times at the various nodes. Moreover, the numbers of jobs occupying the various nodes and classes are independent, Poisson distributed random variables.

In the case of homogeneous input, the time-dependent distribution of state given by (6) can be approximated by the asymptotic distribution given in (7) when t is large enough. The closeness of this approximation can be gauged as follows. Let $\pi_t(C)$ denote the right side of (6) and $\pi(C)$ the right side of (7) and then define

$$d(\pi_t,\pi) \equiv \sum_C |\pi_t(C) - \pi(C)| \ .$$

We can think of $d(\pi_t,\pi)$ as the distance between the distribution of state at time t, namely π_t, and the asymptotic distribution of state π. (In the terminology of measure theory, $d(\pi_t,\pi)$ is the total variation of the signed measure $\pi-\pi_t$.) For the multidimensional Poisson distributions π_t and π, it can be shown (cf. Cinlar [13, pp. 564-565]) that

$$d(\pi_t,\pi) \leq 2 \sum_{j=1}^{N} \sum_{r=1}^{M} [\xi_{rj} - \xi_{rj}(t)] \ .$$

But the summation on the right side of this inequality is

$$\xi - \xi(t) = \lambda \int_t^\infty P\{L > y\}dy \ .$$

By Chebyshev's Inequality (cf. Chung [12, p. 48]), we have $P\{L > y\} \leq E\{L^n\}/y^n$, whence

$$\int_t^\infty P\{L > y\}dy \leq E\{L^n\} \int_t^\infty y^{-n}dy = E\{L^n\}/(n-1)t^{n-1}$$

for $n=2,3,\ldots$, where

$$E\{L^n\} = \sum_{j=1}^{N} \sum_{r=1}^{M} b_{rj} E\{(L_{rj})^n\}$$

and the moments of the variables $\{L_{rj}\}$ can be computed recursively using (5). Thus, if

$$\mu_n(t) = [\lambda/(n-1)t^{n-1}]\ E\{L^n\}$$

then

$$\xi - \mu_n(t) \leq \xi(t) \leq \xi$$

and

$$d(\pi_t,\pi) \leq 2\ \mu_n(t)\ . \tag{8}$$

Similar results hold for the marginal distributions of π_t and π. Indeed, if B is a subset of $\{(r,j)\colon r=1,\ldots,M$ and $j=1,\ldots,N\}$, and π_{Bt} and π_B the corresponding marginal distributions of π_t and π, respectively, then

$$\sum_B \xi_{rj} - \mu_{Bn}(t) \leq \sum_B \xi_{rj}(t) \leq \sum_B \xi_{rj}$$

and

$$d(\pi_{Bt},\pi_B) \leq 2\ \mu_{Bn}(t)$$

where

$$\mu_{Bn}(t) = [\lambda/(n-1)t^{n-1}] \sum_B b_{rj} E\{(L_{rj})^n\}\ .$$

Moreover, if X has a Poisson distribution with mean β then (cf. Hoel et al [30, p. 107]

$$P\{X \leq \beta/2\} \leq (\sqrt{2/e})^{\beta}$$

and

$$P\{X \geq 2\beta\} \leq (e/4)^{\beta}\ .$$

Letting

$$\alpha_{Bn}(t) = \sum_{B} \xi_{rj} - \mu_{Bn}(t)$$

it then follows that

$$\begin{aligned} & P\{C_{rj}(t) \leq \xi_{rj}(t)/2 \text{ for all } (r,j) \text{ in } B\} \\ & = \prod_{B} P\{C_{rj}(t) \leq \xi_{rj}(t)/2\} \\ & \leq \prod_{B} (\sqrt{2/e})^{\xi_{rj}(t)} \\ & \leq (\sqrt{2/e})^{\alpha_{Bn}(t)} \end{aligned} \tag{9}$$

and

$$P\{C_{rj}(t) \geq 2\xi_{rj}(t) \text{ for all } (r,j) \text{ in } B\} \leq (e/4)^{\alpha_{Bn}(t)} \tag{10}$$

Finally, if $\mathscr{C}^* = \{C^*_{rj}\}$ is a random vector having distribution π, then letting $t \to \infty$ in (9) and (10), we see that

$$P\{C^*_{rj} \leq \xi_{rj}/2 \text{ for all } (r,j)\} \leq (\sqrt{2/e})^{\xi} \tag{11}$$

and

$$P\{C^*_{rj} \geq 2\xi_{rj} \text{ for all } (r,j)\} \leq (e/4)^{\xi} . \tag{12}$$

The preceding results suggest that, if possible, the resources placed at node j ($1 \leq j \leq N$) should be adequate to simultaneously process at least $2\xi_{rj}$ jobs in class r ($1 \leq r \leq M$), whenever the external input process is Poisson with rate λ. More generally, if a fixed amount of processing capacity, say P_o, is available to the system as a whole, then the portion of that capacity placed at node j should be $(\sum_{r=1}^{M} \xi_{rj}/\xi)P_o$.

2.4 EXTENSIONS OF SDP

The spatial dynamic programming methodology was used to solve a deterministic software allocation problem. In this section we consider two extensions of the methodology in order to achieve a more realistic model. First, the method is extended to allocate files as well as tasks. Second, allocation with the possibility of hardware failures is considered.

Previously, communication was assumed to occur directly between tasks. In reality, when a task finishes executing it may write its results to a file. Another task reads the file, then uses the data to perform its job. The file must be stored in a memory, and thus requires a certain amount of space. The allocation of files must be done along with the allocation of tasks.

The extension in this case is quite simple. We consider a file to be a "task" which requires memory space but has no instructions. It communicates with all tasks that write to it or read from it. A certain amount of memory is needed for each file. We can then allocate the files along with the tasks in the SDP algorithm.

The second extension, involving hardware failures, is far more complex. For clarity, the problem will be simplified to that of minimizing the finishing time of all tasks, subject to memory constraints. Other constraints, such as communication and task execution time, can easily be incorporated.

We will assume that each hardware element (processor with dedicated memory) has an independent probability of failure, say p. This is a rather strong assumption, but it greatly reduces the amount of computation required. Processor dependent failure probabilities can be accommodated, with a consequent increase in computational load.

Important tasks should be assigned to more than one processor. If a processor fails, the tasks can still execute on another processor to which they have been allocated. If p is small, it is unlikely that more than one processor will fail. As a result, we will assume that no task is assigned to more than two processors. If a task is assigned to two, then one is designated the primary assignment and the other the secondary assignment. The task runs on the primary processor unless that processor fails, in which case the task runs on the secondary processor.

The optimization criterion should be based on the performance of the system. If there are no hardware failures, we can again minimize the task finishing time. Without memory constraints, every task would be allocated to two processors and would be certain of executing. However, if some tasks are allocated to only one processor, these may not execute. This should degrade performance somewhat.

Define d_i to be a penalty which is incurred if task i is not executed. This value can be high for critical tasks, low for less important ones. It can be in any units meaningful to the decision-maker. If tasks $i_1, i_2, \ldots, i_k$ are not executed, the total penalty is some function $f(d_{i_1}, d_{i_2}, \ldots, d_{i_k})$.

Now we need to combine the two attributes of time and penalty. For a given allocation and a given realization, let T_j be the completion time of processor j and D the set of non-executed tasks. We can then postulate a multi-attribute utility function U, of the form $U(\max_j T_j, f(D))$ which is decreasing in both arguments. The objective is to find the allocation which maximizes the expected utility.

This problem cannot be solved by SDP. The problem is that both the maximum time and the penalty are global measures. The combination of the two given by the utility function cannot be broken down into the stage-by-stage optimization problems needed by SDP.

However, suppose the performance criterion is given instead by $U(E(\max_j T_j), E(f(D)))$, where U is some nonlinear function and E is the expectation operator. Now the problem is separable, and can be solved by SDP. The only requirement is that the function f be of the nested form needed for SDP, in which the function value at the current node depends only on the variables at the current node and the function value at the previous node. Examples of such functions are the sum, product, or maximum of the penalties.

Figure 2.10 illustrates the situation. There are a finite number of allocations, each of which incurs an expected completion time and an expected penalty. Indifference curves can be drawn using the U function to give the tradeoff between time and penalty. The allocation on the lowest indifference curve is the optimal solution.

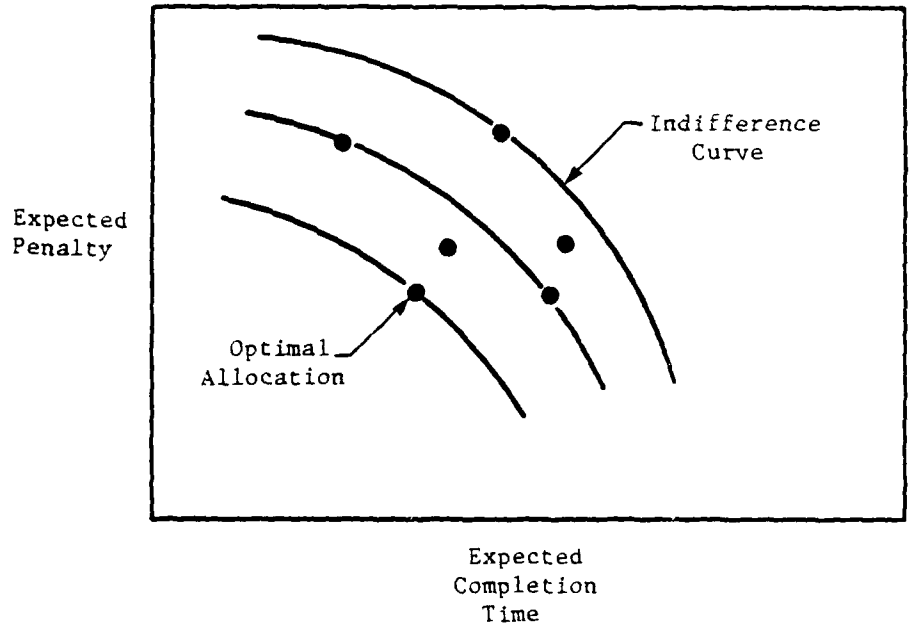


Figure 2.10. Time/Penalty Tradeoff

In the deterministic case, cf. [17], each decision variable x_{ij} took one of two values, 0 or 1, depending on whether or not task i was allocated to processor j. Now we need to distinguish between primary and secondary assignments. The decision variables will now take one or four values, which for simplicity will be denoted 0, 1, 2, and 3.

$x_{ij} = 0$ if task i is never assigned to processor j.

$= 1$ if processor j is the primary assignment for task i, and there is a secondary assignment.

$= 2$ if processor j is the secondary assignment for task i.

$= 3$ if processor j is the primary assignment for task i, and there is no secondary assignment.

The reason for distinguishing between the values 1 and 3 is that if processor j fails and $x_{ij} = 1$ no penalty is incurred, but if $x_{ij} = 3$ a penalty is incurred.

Define S_{ij}^k as the set of tasks for which $x_{ij} = k$. If processor j does not fail, its expected finishing time is

$$T_j = \sum_{i \in S_{ij}^1} t_{ij} + \sum_{i \in S_{ij}^3} t_{ij} + p \sum_{i \in S_{ij}^2} t_{ij}$$

where t_{ij} is the execution time of task i on processor j. The expected penalty is $pf(S_{ij}^3)$.

The allocation problem posed above still cannot be directly solved by SDP. Instead we solve the problem

Minimize $\max_j T_j$

Subject to $f(D) \leq Z^*$

and memory constraints

for several values of Z^*. For each value of Z^*, call the solution to this problem T^*. Several pairs (T^*, Z^*) are obtained, and can be evaluated using the U function. The allocation which gives the optimal pair is the solution.

In order to use SDP, another state variable Z is appended. Z is discretized using relevant values of the expected penalty. At each stage the problem of minimizing the finishing time is solved, but subject to the penalty being no more than each value of Z. At the final stage the values of Z become those of Z^*, and the comparison of solutions can be done.

The computations needed for this problem are significantly more than for the deterministic problem. To begin with, the fact that the number of values for each decision variable is increased from 2 to 4 essentially squares the number of computations at each stage. The addition of another state variable (the penalty level) also increases the computations. However, if the SDP procedure is used only in the design process to find an optimal allocation, this may not be too burdensome.

SECTION III

OPTIMAL ARCHITECTURES FOR DISTRIBUTED SYSTEMS

As stated in the introduction, SCT's long-run goal is to develop a methodology to simultaneously solve the software allocation and hardware design problems. Our first step towards this goal is to look at the two problems separately. A method for integrating the two problems will be developed later.

This section on optimal architectures is organized as follows. Section 3.1 presents an overview of the problem. Section 3.2 describes a computer program which was developed as a tool to evaluate candidate architectures. Section 3.3 looks at a particularly important problem which involves small interprocessor distances. Section 3.4 examines several problems related to network reliability. The final section (3.5) discusses the topic of bus connection networks, which are a more general and more realistic way of describing a distributed computer system.

3.1 PROBLEM OVERVIEW

In this section we consider the problem of how to optimally design a distributed system architecture. Before looking at various architectures, though, we need to define the system's building blocks. The main system elements are a set of processors. Each processor has its own dedicated memory. The interprocessor communication is handled via a specified communication structure. In Sections 3.1-3.4 this structure consists of a set of bidirectional links between processors (although unidirectional links will also be mentioned). Section 3.5 looks at a more general communication structure, in which groups of processors are connected to buses.

The communication mechanism will be assumed to be a message-passing protocol. That is, if processor A needs to send a message to processor B, the message is sent through a sequence of processors until it reaches B.

Either a routing algorithm or a look-up table can be used to determine this sequence. In a local network (such as an avionic system), the time to send the message along each link is insignificant compared to the delay incurred by having to go through several processors. Thus the delay incurred can be assumed to be proportional to the number of intermediate processors along the route between the given processors.

For a real system, the delay depends on both the specific hardware used and the operating system. As a result, this model only pertains to some systems. For example, in the case of multiple processors connected to a linear bus, the delay may be proportional to the total number of processors connected.

A graph-theoretic model has been used to analyze various possible architectures. Such a model uses only the network topology, or how the processors are connected, to evaluate how good the network is. In reality, there are many other considerations, primarily involving the particular software that is going to be used on the system. Nevertheless, the topological analysis can tell us quite a bit about the network performance.

The following graph theory terms will be frequently used in this chapter:

Note: a graphical representation of a processor, usually shown as a dot (·).

Edge: a line segment connecting two nodes, representing a bidirectional communication link between two processors.

Degree of a node: the number of edges connected to a node.

Degree of a graph: the maximum of the degrees of the nodes in the graph.

Regular graph: a graph in which every node has the same degree.

Distance between two nodes: the minimum number of edges between two nodes.

Diameter of a graph: the maximum distance between all pairs of nodes.

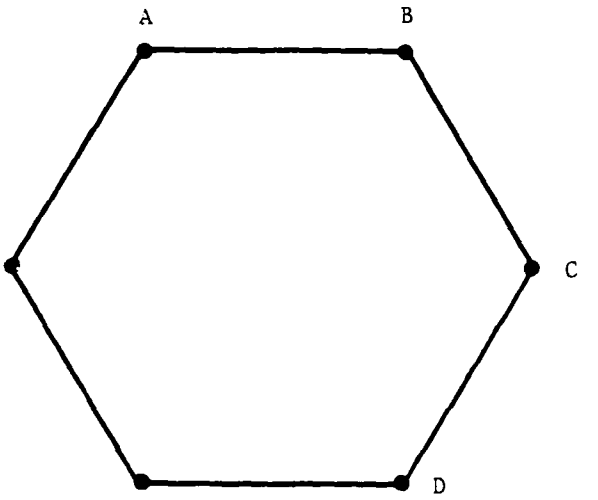


Figure 3.1. Ring Network

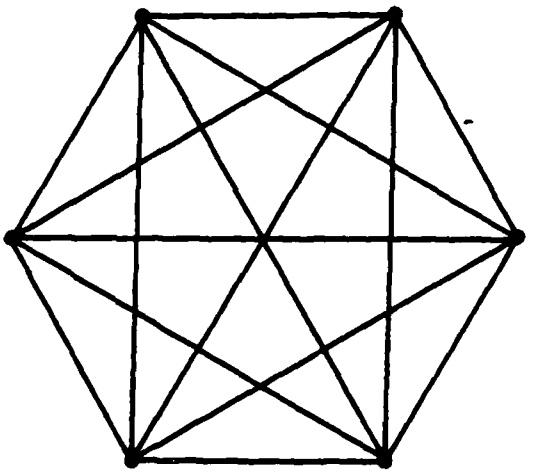


Figure 3.2. Fully Connected Network

To show how these terms are used, consider a few possible distributed architectures, shown in Figures 3.1 - 3.3. Figure 3.1 shows a ring architecture on 6 nodes. There are 6 edges in the graph. Each node has degree 2, so the graph is regular and has degree 2. The distance between nodes A and B is 1, between A and C is 2, and between A and D is 3. The diameter of the graph is 3.

Figure 3.2 shows a fully connected architecture on 6 nodes. Every node has degree 5, so the graph is regular. Since every pair of nodes is connected by a link, the distance between each pair is 1. Thus the graph's diameter is 1.

Figure 3.3 shows a hierarchical architecture with 7 nodes. This graph is not regular, since the top node has degree 2, the next two nodes have degree 3, and the bottom nodes have degree 1. The diameter is 4.

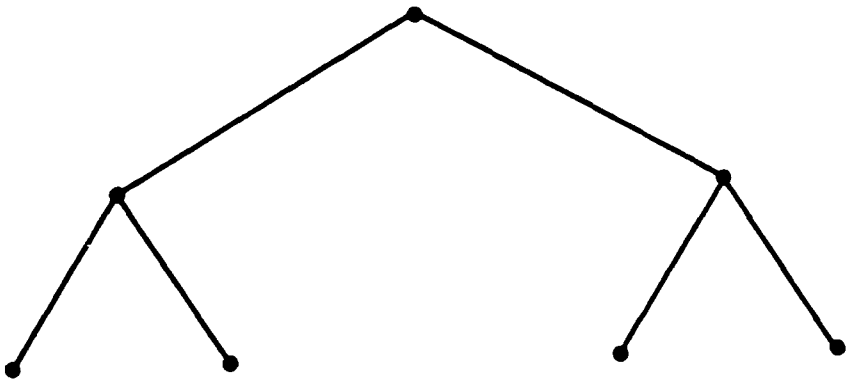


Figure 3.3. Hierarchical Network

In designing a processor interconnection network, there are many different factors to consider. These factors include the following:

1) The number of connections per processor should be small. This is motivated by feasibility and cost; processors with a large number of ports may not be available, or may be very costly.

2) The distance between processors should be small. In a message-passing network, the delay time for a message is approximately proportional to the number of links which must be traveled over between processors. The minimization of delays should be one of the objectives of the network design.

3) No processor or link should be on a high proportion of the shortest paths between processors. Such a configuration would cause queueing delays, which would degrade the performance.

4) The network should be highly reliable. If a processor or link fails, the network's performance should not be seriously worsened.

These various factors are often in conflict. For example, if we allow each processor to be connected to only two others, then a ring architecture (Figure 3.1) is the best possible. With n nodes, the diameter of this graph is n/2. A fully connected network (Figure 3.2) has a diameter of only one, but requires each of n processors to be connected to n - 1 others.

Figure 3.4 illustrates the tradeoff between the two factors of degree and diameter in the case where we have 12 nodes. As the allowable degree increases, the possible diameter decreases. Some graphs which correspond to these values are shown in Figure 3.5. The best graph among these depends upon the desired cost/performance tradeoff.

Suppose the cost of a network with n processors depends solely on the number of links in the network. Let us assume that each processor can accommodate as many as n-1 links. If the links are very expensive, the best network will have as few links as possible. Assuming that the network

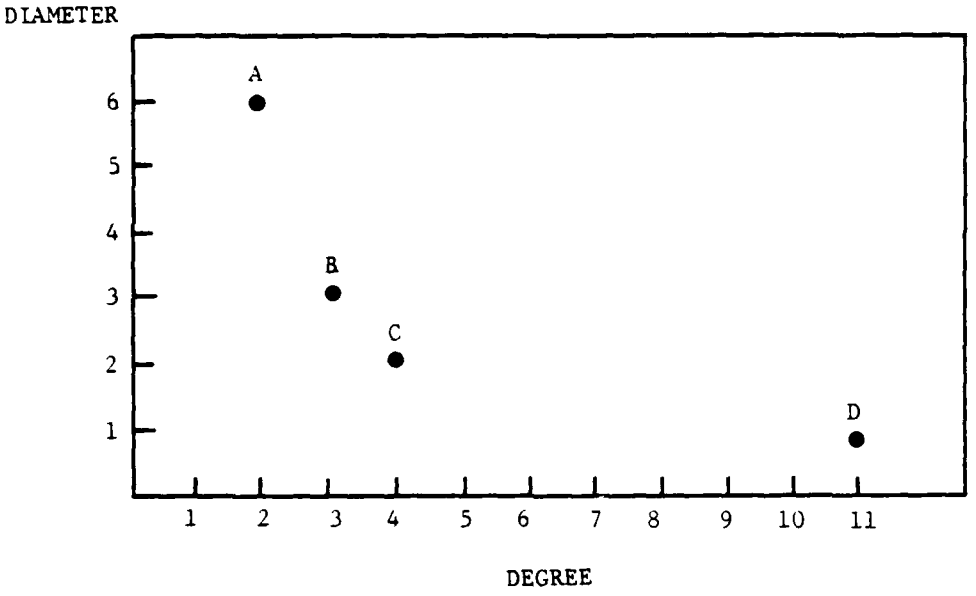


Figure 3.4. Degree/Diameter Tradeoff for 12-Node Graphs

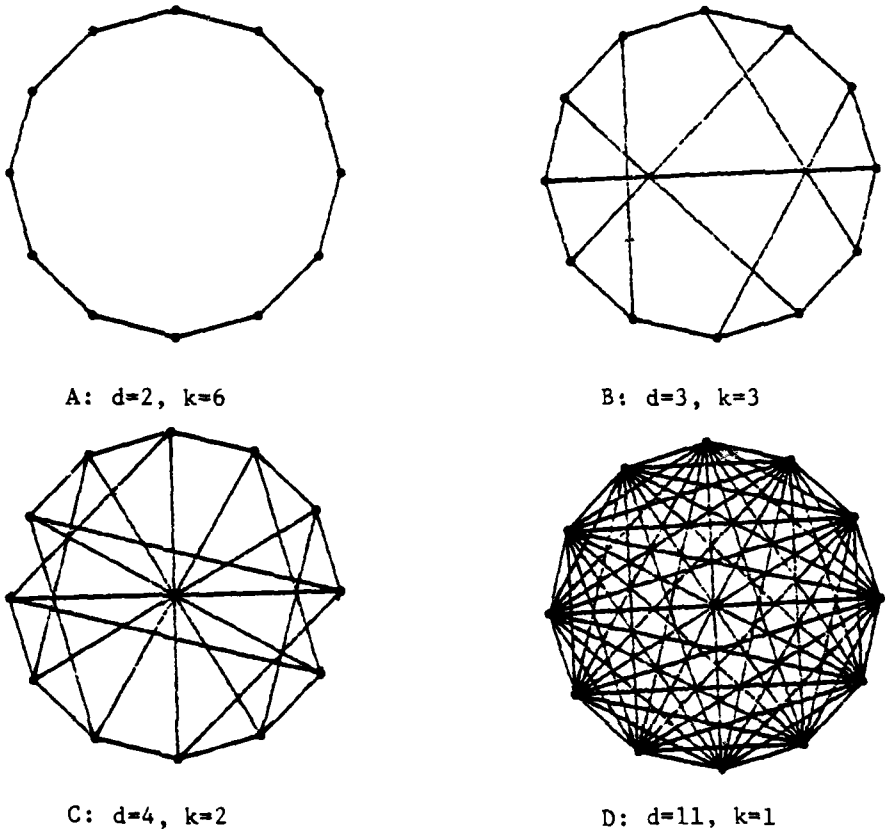


Figure 3.5. Possible 12-Node Graphs with Degree d, Diameter k

must be connected (there must be a path between every pair of nodes), the best network would be a tree, such as that in Figure 3.3 The tree with the smallest diameter is a star, such as the one in Figure 3.6. A major problem with this type of network is that all messages between processors must go through the central processor. Furthermore, if the central processor fails the network becomes disconnected.

Clearly there are many graph theory questions which are of interest in the design of distributed computer systems. The next section describes a tool which was developed in order to look at some of these questions.

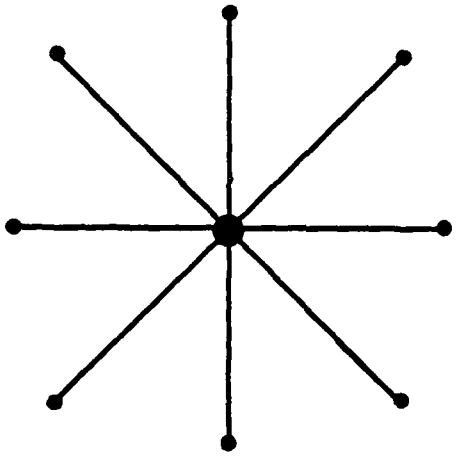


Figure 3.6. Star Network

3.2 A NETWORK EVALUATION TOOL

In order to evaluate a proposed network architecture, a computer program called NETEV has been developed. The purpose of this program is to serve as a design tool. Parameters which characterize the network are inputs to the program. The outputs specify the performance of the network, both with and without hardware failures. A listing of this program appears as Appendix B of this report. It is written in FORTRAN, for a DEC VAX 11/780.

The program inputs are:

-- Number of processors
-- Number of links between processors
-- Where the links are
-- "Distance" of links (which need not be one)

The program outputs are:

-- Maximum inter-processor distance (diameter)
-- Average inter-processor distance
-- Expected diameter, given a single processor failure
-- Expected average distance, given a single processor failure
-- Expected diameter, given a single link failure
-- Expected average distance, given a single link failure
-- Fraction of links which can fail and leave the network connected
-- Average fraction of connected processors, given a single link failure

The code assumes that the links are bidirectional. However, the algorithm does not require this, and a simple modification of the code would allow unidirectional links.

The first two outputs characterize the network performance if there are no hardware failures. To analyze the effect of failures, we assume a model in which the probability that a processor or link fails is small, so that

the probability of more than one failure can be safely neglected. Furthermore, we assume that every processor fails with equal probability, and that every link fails with equal probability. The program can be easily modified so that the probability that each processor or link fails is an input.

There are a number of different measures of the effect of hardware failures on network performance. We can analyze how the diameter or average distance is increased as a result of a node or link failure. However, for many networks (such as hierarchical ones) these values will be infinity, and so little information is gained on how reliable the network is. The last two measures attempt to rectify this problem. In some networks it may be important for all processors to communicate. Thus the fraction of links which can fail and yet leave the network connected is relevant. In a ring architecture any link can fail and the network will still be connected; while in a hierarchical architecture if any link fails the network will be disconnected. However, perhaps not all pairs of processors are required to communicate. As a result, we may want to know about what fraction of the processors can communicate, given a link failure.

The heart of the program is the calculation of the distance between all pairs of processors. This is done by using the Floyd-Warshall algorithm [24], which is an efficient method for finding the distance between all pairs of nodes in a graph. This algorithm is used on the network for each possible node or link failure.

As an example of how the method works, consider the two graphs shown in Figure 3.7. These graphs are examples of the chordal rings of Arden and Lee [2]. Notice that in the left graph each chord (the links inside of the ring) subtends 5 nodes, while in the right graph each chord subtends 7 nodes. To determine which of these is optimal we can use NETEV to calculate the maximum distance and average distance. Assuming that each link has length 1, the two graphs have identical diameters (4) and average distances (2.5263). However, suppose we can use links with a different

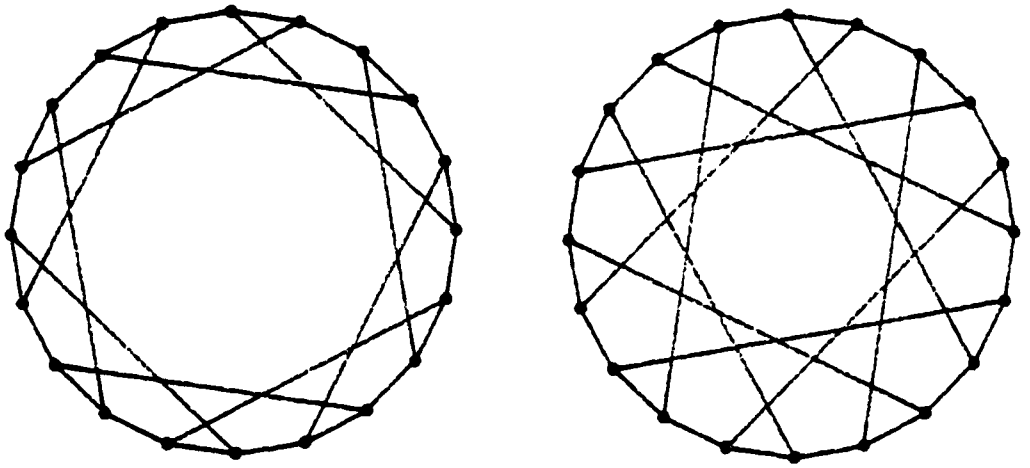


Figure 3.7. Two Chordal Rings

bandwidth for the chords. Using NETEV, we find that if the bandwidth of the chord links is greater than that of the ring links, then the left graph is optimal; while if it is less, the right graph is optimal. This result corresponds to our intuition. If the chord bandwidth is low, the chords must subtend many nodes in order to be on a shortest path; while if the chord bandwidth is high shorter jumps can be used. NETEV can also be used to compare the performance with failures, but in this case the two graphs perform identically.

NETEV allows us to compare networks using many different criteria. If we limit ourselves to only a few criteria, some interesting quantitative results can be obtained, as shown in the next section.

3.3 MINIMAL DISTANCE PROBLEM

Two of the fundamental quantities related to a distributed computer system are the cost and the performance. From a graph-theory viewpoint, the cost (for a fixed number of nodes) may be represented by the number of edges. The measurement of performance is somewhat more difficult.

Without knowledge of the software to be used on the system, we need to be concerned about the distance between all pairs of nodes. It is difficult to work with the whole set of distances, so various statistics about the distances are used. Two commonly used statistics are the average distance and the maximum distance, which equals the network diameter.

Our work has principally focused on the diameter, for several reasons:

- The diameter characterizes the worst-case situation, which is likely to be the one of most concern.

- Networks with small diameters tend to have small average distances, while the converse is not necessarily true.

- The diameter is quicker to use in computations. Given two graphs, if the second has a pair of nodes which is further apart than the diameter of the first graph, then the second graph has a larger diameter. To compare average distances, all of the distances in the second graph would have to be calculated.

Many interesting questions arise from the cost/performance tradeoff. For example, given a number of nodes n and an allowed diameter k, what graph has the fewest edges? If $k = 1$, we must have a complete graph (Figure 3.2) with $n(n-1)/2$ edges. If $k = 2$, then a star network (Figure 3.6) can be used, with $n-1$ edges. For any larger allowed diameter, we must still have $n-1$ edges, in order for the network to remain connected and the diameter to remain finite.

A problem with both the complete graph and the star graph is that at least one node must have degree $n-1$. Real processors can be connected to only a limited number of other processors. Thus there should be a bound on the degree of a node.

The following problem was posed by Bollobas [9]: Given a number of nodes n, a maximal degree d, and a maximal diameter k find a graph with the fewest edges. Unfortunately, few results for this problem can be obtained except in limiting cases. The maximum number of edges under these conditions is nd/2, which would occur if every node had degree d. Will this number of edges be sufficient to produce a graph with the desired diameter? If k is too small, it will not be. However, if $k \geq n/2$, for any $d \geq 2$ there will be a graph (a ring). Thus there will be some minimal diameter for which there exists a graph with n nodes and degree d. We seek this minimal diameter, for it gives us a bound on the network performance. This minimal diameter network is likely to be a regular graph, and as such it may have more links than necessary for a real system. However, to design a real system we could start with such a graph and then remove unnecessary links.

Thus the problem we are interested in is: Given a number of nodes n, and a maximum degree d, find a graph with the minimum diameter k. In general, as the number of nodes increases the minimum diameter increases. Thus we can consider a dual problem: Given a maximum degree d and a diameter k, find a graph with the maximum number of nodes. Of course, when designing a system we are not really going to choose an alternative with the most processors. Rather, the solution to the dual problem will say that for a degree d and diameter k, we can have as many as n nodes. If we want to construct a network with $n' < n$ nodes, we can probably find one with diameter k or less.

The maximum number of nodes in a graph with degree d and diameter k can be denoted n(d,k). An upper bound on n(d,k) is easily calculated. From any given node at most d nodes can be reached in a distance of one and, for $j > 1$, at most $d(d-1)^{j-1}$ nodes can be reached in a distance of j. Thus

$$n(d,k) \leq 1 + d + \ldots + d(d-1)^{k-1} = \frac{d(d-1)^k - 2}{d - 2} \qquad (1)$$

Expression 1 is called the Moore bound, and any graph which has that number of nodes is called a Moore graph. Most Moore graphs fall into two classes: 1) rings with an odd number of nodes, where $d = 2$; 2) fully connected networks, where $k = 1$. In [31] it was shown that for $k = 2$ there are only a few other Moore graphs: the Petersen graph (Figure 3.8) where $d = 3$; the Hoffman-Singleton graph, where $d = 7$; and possibly a graph with $d = 57$. In [5] and [15] it was shown that there are no other Moore graphs. In [6] it was shown that except for the square there are no graphs with a number of nodes equal to one less than the Moore bound. No better upper bound on $n(d,k)$ has been established.

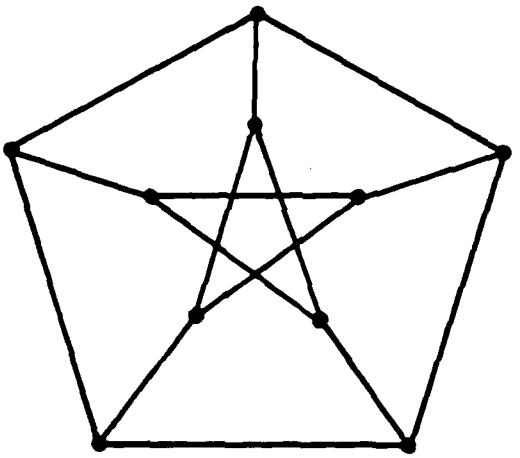


Figure 3.8. Petersen Graph

We shall denote by $b(d,k)$ a lower bound on $n(d,k)$. Values of $b(d,k)$ can be obtained by exhibiting a graph with degree d, diameter k, and $b(d,k)$ nodes. A number of authors have written papers on finding improved values of $b(d,k)$. These authors include Elspas [21], Akers [1],

Friedman [25], Korn [36], Storwick [44], Arden and Lee [3], and Leland et al [37]. In each of these papers, a new construction method is given. Some of these methods are shown in Figures 3.9 - 3.11. Figure 3.9 shows a "star polygon" which was used by Elspas. In Figure 3.10 a "hinging" graph is shown. It can be thought of as three hierarchical graphs, with the nodes on the lowest levels joined together. This method was used by Friedman and Korn, and a generalization of it was used by Storwick. Arden and Lee used a "multi-tree structured network" approach, an example of which is shown in Figure 3.11. Leland used a heuristic method to construct his graphs, which are the largest published graphs for small degrees and diameters.

SCT has discovered a new class of graphs which are larger than those of the other authors for many degrees and diameters. These are called chordal ring graphs, and are a generalization of a structure proposed by Arden and Lee [2]. Two examples of Arden and Lee's chordal rings are shown in the previous section, Figure 3.7. In their structure, every node has degree 3. The graph begins as a ring on n nodes, then chords are drawn connecting additional pairs of nodes. Each odd node is connected to the even node which is w nodes ahead of it on the ring, where w is some specified odd number.

The generalization is twofold. First, more complex chordal connection schemes are used. The Arden and Lee rings can be thought of as having a pattern of length two, in that every other node is connected to the node w nodes ahead of it. This can be generalized to larger pattern lengths, or orders. Figure 3.12 shows a chordal ring of order three, in which every third node is connected to the node across the ring from it, while the other nodes are connected to nodes which are 8 nodes either ahead of or behind them. This graph has diameter 4.

For a given number of nodes, finding the optimal connection scheme usually requires a computer search. For each feasible order (the order must divide the number of nodes), all combinations of chord lengths must be checked. Letting r be the order and n the number of nodes, the number of combinations has the form $c(r)\cdot(n/r)^{r/2}$, where $c(r)$ is a

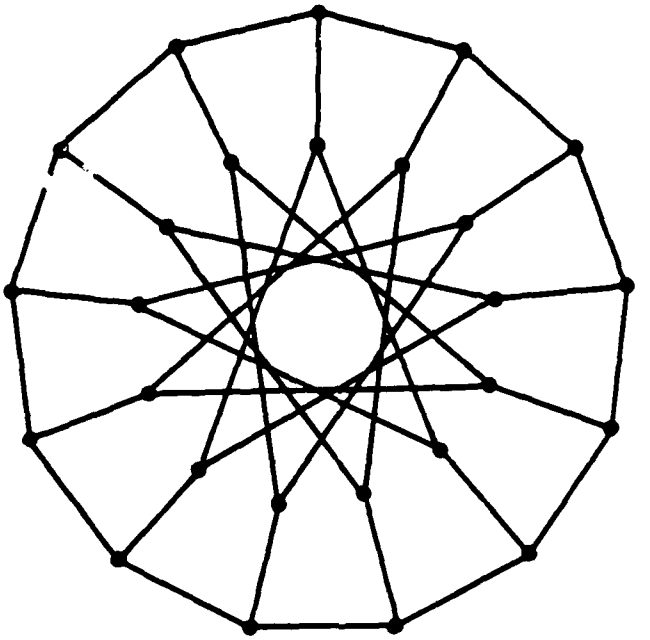


Figure 3.9. Star Polygon

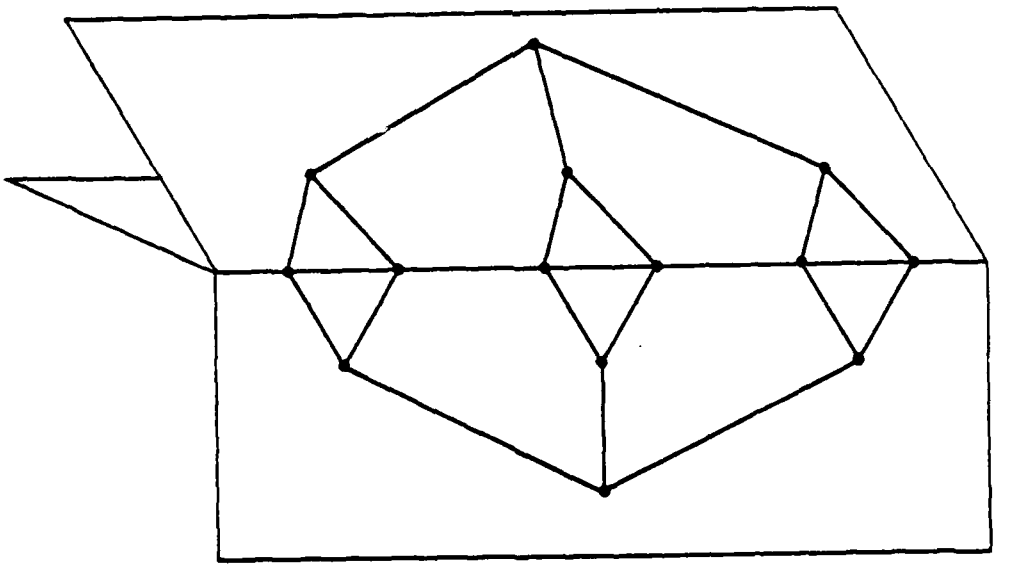


Figure 3.10. Hinging Graph

function which grows like $r!$. (See [20] for this derivation). As either n or r grows large, checking all possibilities becomes infeasible. For most of the results given here, a random search method was used, examining a fixed number (about 1000) of possibilities.

The second generalization is that larger degrees can be used in chordal rings. Figure 3.13 shows an example of a chordal ring with degree 4, diameter 3, and 36 nodes. While the computations for finding optimal chordal rings are even more burdensome with larger degrees, those found by the random search method are better than graphs constructed by any other method.

For the smallest cases which are unsolved (degree 3, diameters 4 and 5), heuristic methods were able to produce larger graphs than in the literature. The heuristics, however, were based on chordal rings. Figure 3.14 shows the 38 node diameter 4 graph, while Figure 3.15 shows the 60 node, diameter 5 graph. This size of a system is probably at the upper end of the size of avionic systems in the near future.

The status of the problem of maximizing $b(d,k)$ is shown in Table 3.1, for degrees and diameters not exceeding 7. For some cases (those circled), the largest possible graph has been found. In the other cases, the value of $n(d,k)$ is an open question. For those cases in which there are two numbers, the top number is the best published result while the bottom number is the best result obtained by SCT. Notice that in almost all cases we have been able to obtain better results.

In order to construct larger networks (with several thousand nodes), different methods must be used. While such networks are unlikely to be used for avionic systems in the near future, they are a natural outgrowth of the work on small networks. Furthermore, with the increasing miniaturization of processors, such networks may be feasible in a few decades.

The paper by Imase and Itoh [33] describes a particularly interesting network based on de Bruijn sequences [16]. For degree d and diameter k, these networks have $(d/2)^k$ nodes, which for $d \geq 6$ and sufficiently

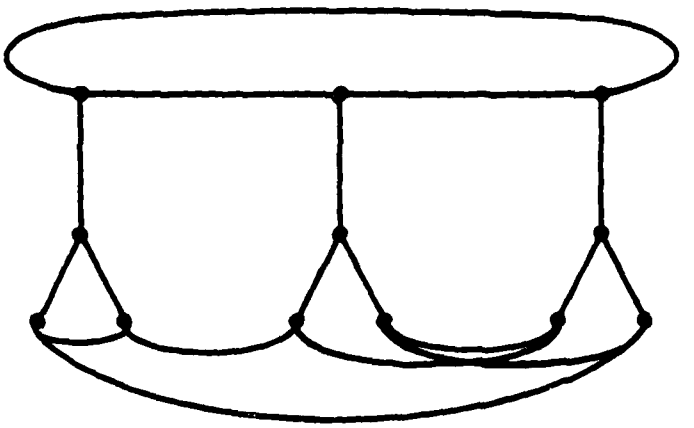


Figure 3.11. Multi-Tree Structured Network

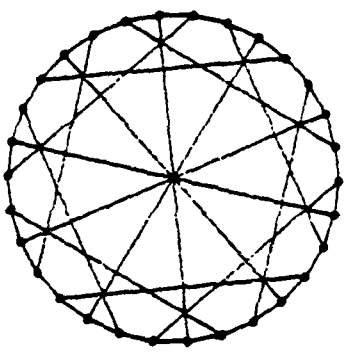


Figure 3.12 A Generalized Chordal Ring

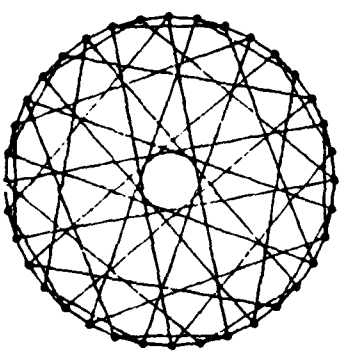


Figure 3.13. A Degree 4 Generalized Chordal Ring

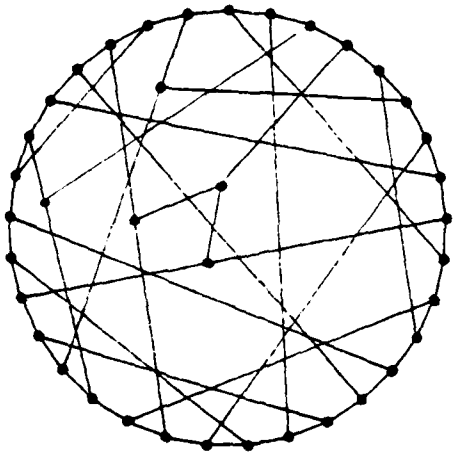


Figure 3.14. 38 Node, Degree 3, Diameter 4 Graph

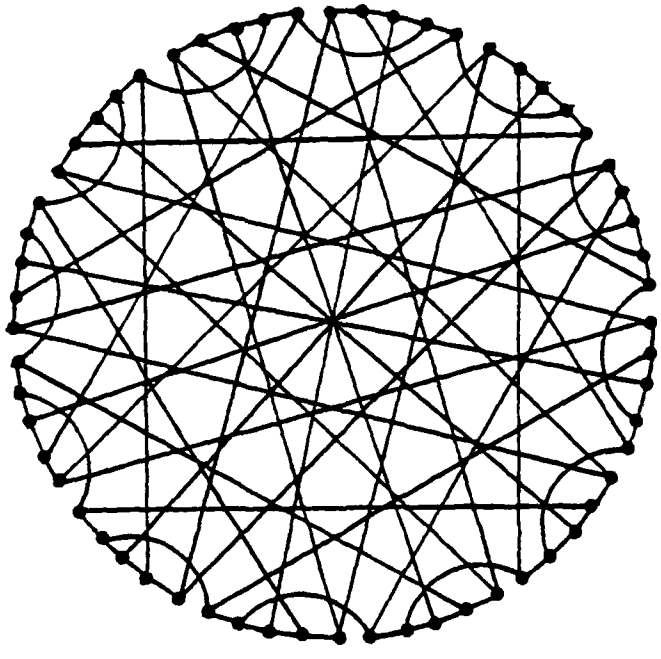


Figure 3.15. 60 Node, Degree 3, Diameter 5 Graph

LARGEST KNOWN GRAPHS FOR DEGREES AND DIAMETERS BETWEEN 1 AND 7

DEGREE \ DIAMETER	1	2	3	4	5	6	7
1	(2)						
2	(3)	(5)	(7)	(9)	(11)	(13)	(15)
3	(4)	(10)	(20)	34 38	56 60	84 100	122 180
4	(5)	(15)	35 36	67 92	134 188	261 378	425 856
5	(6)	(24)	48 60	126 164	262 400	505 1014	1260 2604
6	(7)	31	65 94	164 284	600 820	1152 2604	2520 7000
7	(8)	(50)	88 122	252 420	992 1550	2880 5304	4680 8930

Circled Numbers: Graphs have been proven to be maximal.

Where there are two numbers: Top numbers are from [13], bottom numbers are SCT's results.

TABLE 3.1

large k is larger than any other known type of network. These networks have a simple routing algorithm. Furthermore, if a processor failure occurs a new route can easily be calculated [43]. The networks presented by Imase and Itoh can also be unidirectional, in which case they are almost as large as theoretically possible. The routing algorithm, and how to adjust it when there are failures, is given in Appendix C.

The other principal method for constructing large networks is by using graph products, some of which are described in [37] and [18]. Products of various small graphs are used to produce large graphs. Products of the new chordal ring graphs, using the methods in [37], can produce larger graphs than the examples cited in that paper.

While the problem of maximizing the number of nodes in a graph has not been solved, substantial progress has been made. Of course, there are several other factors we are interested in. The issue of network reliability, and the tradeoff between this and short distances for hierarchical graphs, is discussed in the following section.

3.4 NETWORK RELIABILITY

There are many attributes besides short distances that are of interest when designing a network. This section briefly explores one of these attributes, that of network reliability. In particular, we look at the tradeoff between reliability and short distances.

Network reliability can be characterized in many different ways. We will consider a simple model in which each node or edge fails with a known probability. These failures will be assumed to be independent; the dependent case is much harder.

Some of the important measures of reliability are:

- The probability that a particular node pair, or set of node pairs, can communicate.
- The probability that all node pairs can communicate.

- The probability that all nodes can communicate with one particular node.
- The expected number of nodes which can communicate with a particular node.
- The expected number of node pairs which can communicate.

A few of these are similar to the quantities which can be computed using the NETEV program discussed in Section 3.2. However, that program assumed that the failure probabilities were small, so that the probability that more than one hardware element fails could be safely neglected. If this assumption is not valid, the problem becomes much more difficult. Reference [47] presents an algorithm for computing network reliability, and gives several references to other algorithms.

Bollobas [9] discussed a problem which ties together performance, cost, and reliability. In general, the failure of a node or edge increases the graph diameter. The problem he considered was given a number of nodes n, degree d, and diameter k, find the graph with the fewest edges in which the deletion of any node does not increase the diameter beyond $k' \geq k$. Edge deletion instead of node deletion can also be considered. These problems are even more difficult than the similar problem without failures discussed in Section 3.3, and there are even fewer results. Figure 3.16 shows an example of a critical graph. If an edge is deleted, the diameter may increase from 6 to 9.

Another measure of reliability which may be easier to calculate is the number of nodes or edges which can fail (alone), and yet all remaining node pairs can communicate. Again, this may be a relevant measure only if the probability of multiple failures is very small. There is an interesting tradeoff between this measure and having short distances, which we will now examine.

In particular, suppose we have a graph to which we want to add an additional edge in an optimal fashion. Such a step might be an iteration in a network construction algorithm, or it might be a separate item of interest. The optimal place to add it depends upon the function we want to optimize, as can be seen from a simple example. Suppose we have a large number (several dozen) nodes in a linear architecture (Figure 3.17). Let us find the optimal place to add one additional edge, in order to minimize

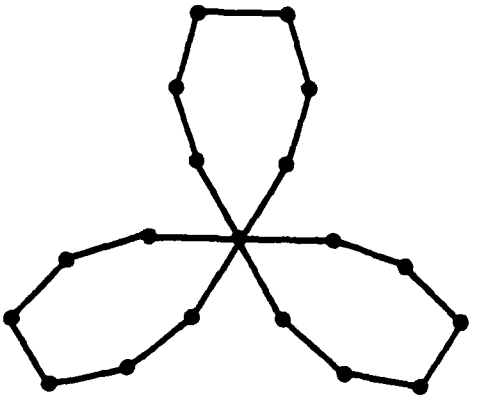


Figure 3.16. Example of a Critical Reliability Graph

Figure 3.17. Linear Architecture

the average interprocessor distance. Intuitively the edge should be symmetric, connecting processors at a fraction f from both ends. Then we can calculate that with n processors the average interprocessor distance is approximately

$$(2f^3/3 + f^2 - f/2 + 1/4)\ n\ .$$

The minimum value of this function occurs at $f = (\sqrt{2} - 1)/2$. Thus the edge should connect processors about 20% from each end.

However, suppose we want to add an edge in order to increase the network's reliability. Let us define the reliability as the fraction of nodes (or edges) which, if they fail, will not disconnect the network.

For the linear architecture example, this is equal to $1 - 2f$. The minimum of this is at $f = 0$, where the result is a ring architecture. Figure 3.18 plots the average distance versus the reliability. It is optimal to be on the right side of the curve, if high reliability and small distances are desirable. Here f is between 0 and $(\sqrt{2} - 1)/2$. However, the best point depends on the tradeoff between these two factors.

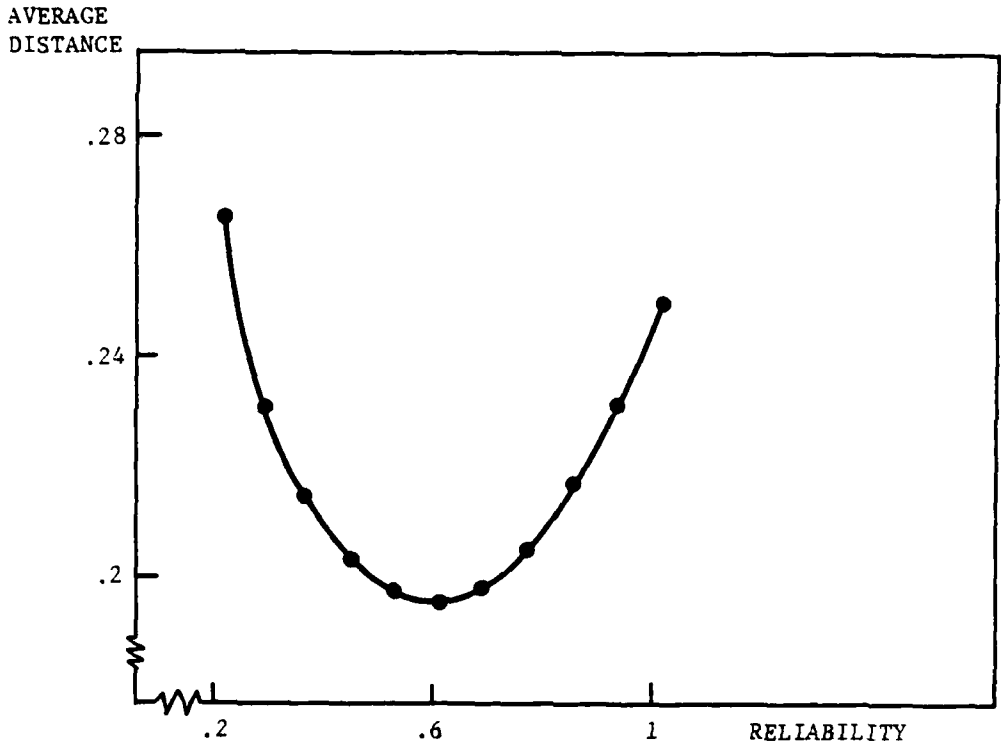


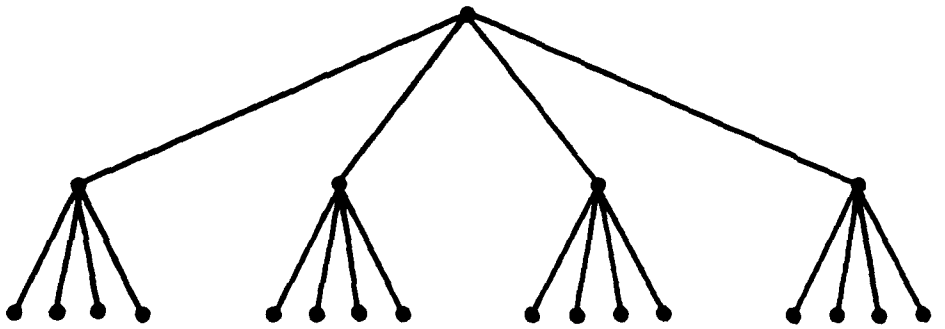
Figure 3.18. Average Distance vs. Reliability for Link Added to Linear Graph

We can also look at the problem of finding the best place to add an edge to a tree, or hierarchical, network, for performance or reliability purposes. Cockayne, Ruskey, and Thomason [14] look at the problem of finding the best place to add an edge to a tree network in order to minimize the average distance. Their algorithm enumerates all node pairs and calculates how much the average distance is improved by adding an edge between

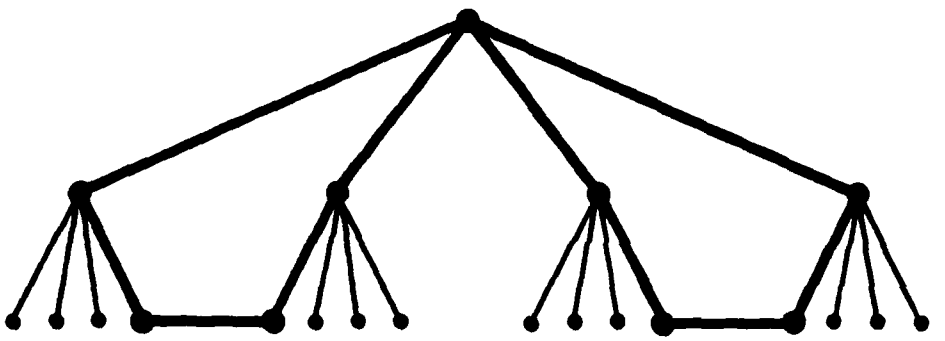
those nodes. This is done in an intelligent manner, so that the number of calculations is only proportional to n^2k, where n is the number of nodes and k is the diameter. A brute force method would require calculations proportional to n^5. Additional details on their method appear in Appendix D.

If we define reliability by the number of nodes or edges which can fail (singly) and keep the network connected, it turns out to be relatively simple to find the best place to add edges to a hierarchical graph. Define a node or an edge to be protected if, when it fails, the network remains connected. For a node of degree d to be protected, the d parts of the graph connected to it must be connected to each other. This requires at least $d-1$ additional edges. For an edge to be protected, it must lie in a cycle (that is, the two nodes which the edge connects must have another path between them).

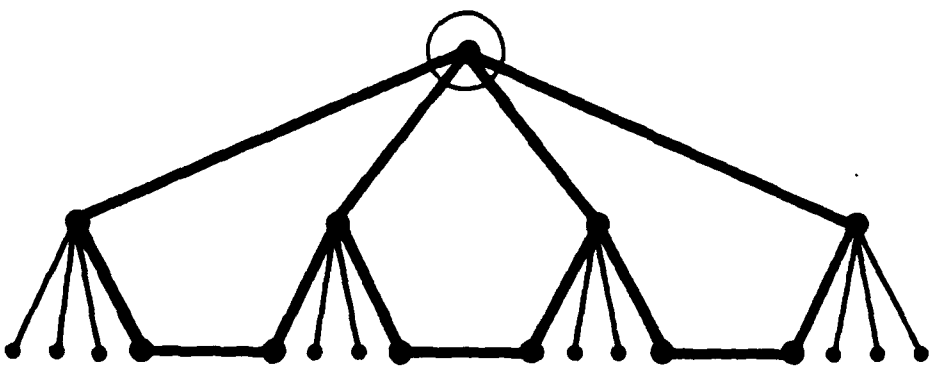
The optimal method can best be illustrated by an example. Figure 3.19a shows a hierarchical graph, which is "regular" in the sense that every node above the bottom row has the same number of successors. The method works only on networks of this type. The basic idea is to protect nodes and edges from the top of the hierarchy to the bottom, protecting as many as possible at each step. In the example, the bottom nodes are originally protected, but no edges are. In order to protect the top node at least 3 edges must be added, while in order to protect each second row node at least 4 edges must be added. The largest cycles which can be created use 4 of the original edges. Figure 3.19b shows how two new edges protect 8 old ones (the dark ones). With one more edge the top node is protected and circled, as well as 2 more edges. This is shown in Figure 3.19c. With 2 more edges, as in Figure 3.19d, two more nodes and 4 more edges are protected. The final two nodes and 6 edges can be protected with 3 new edges, as in Figure 3.19e. Thus 8 edges were needed to protect every hardware element. Clearly fewer are insufficient, as each new edge can protect at most two of the bottom row of edges.



a. Original graph

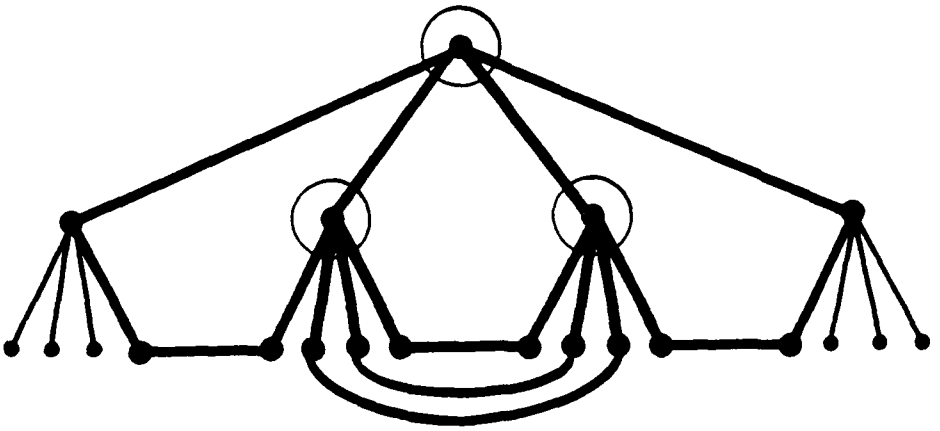


b. Addition of two edges to protect 8 edges

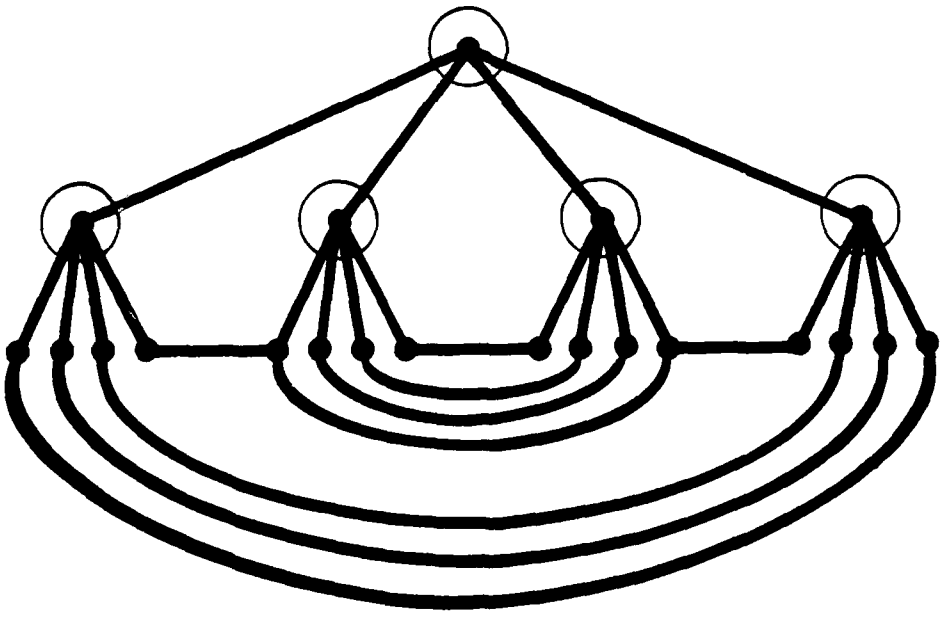


c. One more edge to protect 1 node, 2 edges

Figure 3.19. Additional Edges to Increase Reliability



d. Two more edges to protect 2 nodes, 4 edges



e. Three more edges to protect 2 nodes 6 edges

Figure 3.19 (Continued). Addition of Edges to Increase Reliability

We have seen many types of networks with links which connect two nodes. A more general communication structure may be more faithful to real systems, and may produce better results. This topic is explored in the next section.

3.5 BUS CONNECTION NETWORKS

In some computer systems, the model of having two nodes connected by an edge is not very realistic for the communication mechanism. Instead, several processors may be connected to a bus and may be thought of as being equally distant from each other. At the same time, each processor may be connected to several buses. The result is a bus connection network.

Figure 3.20 illustrates the situation. For processor 1 to communicate with processor 5, a message is sent through bus A to processor 3. This processor then sends the message along bus B to processor 5. Since two buses were used to send the message, we can say that the distance between processors 1 and 5 is 2.

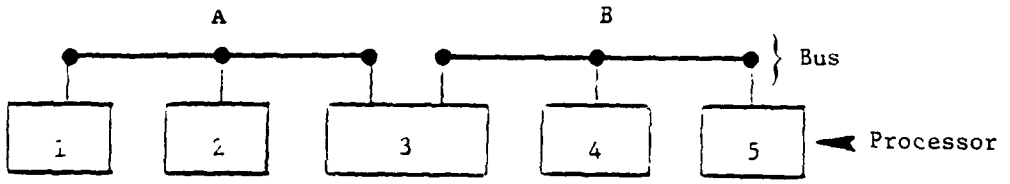


Figure 3.20. Example of a Bus Connection Network

We will use a model discussed by Mickunas [42] for a bus connection model. In this model, each node is incident on a certain number of buses, which we will call its degree. The maximum of the degrees of the nodes will be called the graph's nodal degree. Each bus has a certain number of nodes on it, which will be called the degree of the bus. The maximum of the degrees of the buses will be called the graph's bus degree. For notation, we will denote the nodal degree by d, the bus degree by b, and the diameter by k.

Two nodes will have a distance of one if they are incident on a common bus. In general, the distance between two nodes is the minimum number of buses which must be passed through to get between them.

The standard graphical representation of these networks is to use lines as buses, connecting the nodes which are points. However, these "lines" become complex curves in even very small networks, making any sort of visual analysis impossible. Instead, we will use a representation of the network as a bipartite graph. A bipartite graph has two types of nodes, and nodes of each type are connected only to nodes of the other type. One type of node represents the original nodes, while the other type represents the buses. In the figures in this section nodes are filled-in circles, buses are empty circles. An edge represents a node incident on a bus. Figure 3.21 compares the two representations for a particularly interesting graph with 7 nodes and 7 buses, each with degree 3. The graphs have diameter 1. In the first figure, each side of the triangle, each median, and the circle in the middle represent buses. The node correspondences between the two figures are given by letters. One particular advantage of this new representation is that many results from standard graphs can be used.

A concept analogous to Moore graphs can be defined for bus connection networks. From each node at most d buses can be reached, from which at most $d(b-1)$ nodes can be reached. Thus the maximum number of nodes in a diameter 1 graph is $1 + d(b-1)$. Similarly, in two steps at most $d(d-1)(b-1)^2$ nodes can be reached and, in j steps, at most $d(d-1)^{j-1}(b-1)^j$ nodes can be reached. If we define $n(d,b,k)$ as the maximum number of nodes in a graph with nodal degree d, bus degree b, and diameter k, we have

$$n(d,b,k) \leq 1 + d(b-1) + d(d-1)(b-1)^2 + \ldots + d(d-1)^{k-1}(b-1)^k$$

$$= 1 + d(b-1)\,\frac{(d-1)^k(b-1)^k - 1}{(d-1)(b-1) - 1} \quad .$$

A graph with this number of nodes is called a Moore geometry.

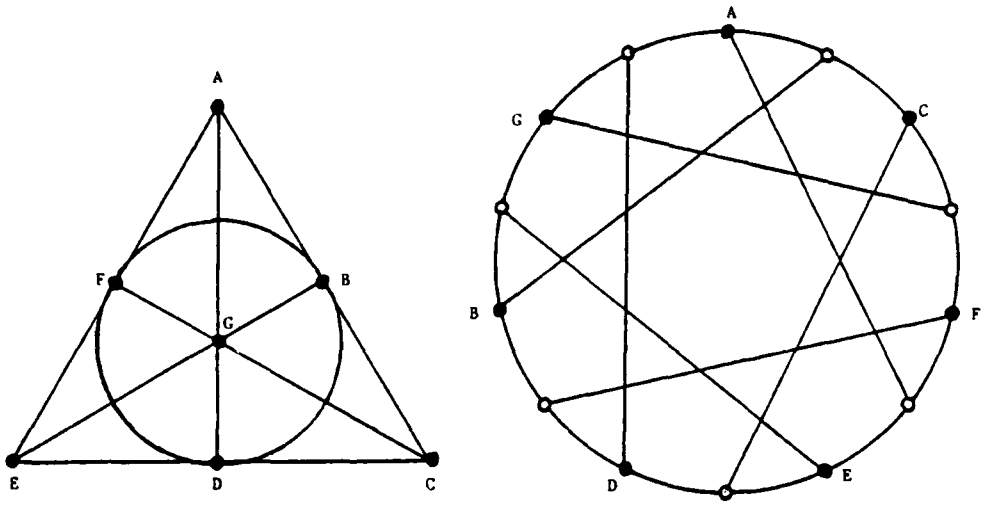


Figure 3.21. Two Representations of 7 Node, Diameter 1 Graph

Since there are very few Moore graphs, it is natural to ask if there are any Moore geometries. Let us first consider the case of $d = 1$. A Moore geometry with diameter 1 has $1 + d(b-1)$ nodes. The simplest case here is where $d = b$ (the node degree and bus degree are the same), so there are $d^2 - d + 1$ nodes, and the same number of buses. Every pair of nodes is on exactly one common bus.

The existence of such a graph depends on the existence of a mathematical object called a finite projective plane, a subject which has been extensively examined. It can be shown that such a plane exists if $d - 1$ is a prime power [45]. In certain other cases it can be proven that a projective plane does not exist [11]. However, the general problem remains unsolved. For values of d between 3 and 10 projective planes exist in all cases except $d = 7$.

Now let us look at the case of $d \neq b$. These graphs are known as balanced incomplete block designs, or BIBDs. Such designs are used in constructing agricultural and biological experiments. There are certain restrictions on the parameters of a BIBD. Since nd/b is the number of buses, this number must be an integer. It can also be shown that we need $d \geq b$ [27]. While these conditions are not sufficient to guarantee the existence of a BIBD, if $d = 3$ or 4 they are sufficient [28]. In addition, if an order $d-1$ projective plane exists, then a BIBD can be derived with the same d, $b = d-1$, and $(d-1)^2$ nodes. Several examples of BIBDs are given in [23].

The question of the existence of Moore geometries with diameters greater than one is more complicated. None are known if $b > 2$. Bose and Dowling [10] give necessary conditions for existence when $k = 2$, although they could find no graphs satisfying those conditions. Fuglister [26] showed that there are no Moore geometries with $k = 3$.

One simple result is that there are no Moore geometries with $d = 2$ and $b > 2$ (if $b = 2$ we get rings). This is an important case since many real microprocessors have two ports. The proof of this is given in Appendix E.

Several construction methods for bus connection networks have been proposed in the literature. These include the hypercube and dual bus hypercube [46], and snowflake and star graphs [22]. These constructions are motivated by their simple structure and easy routing algorithms. If we are interested in small interprocessor distances, however, these simple structures may not be the best.

Consider a problem analogous to the one discussed in section 3.3 for standard graphs. We seek a graph with the maximum number of nodes which has node degree d, bus degree b, and diameter k. To begin with, let us consider some small special cases.

1. $n(1, b, 1) = b$

This is obvious -- it is a single bus of degree b. It is the only trivial case of a Moore geometry with $b > 2$.

2. $n(2, b, 1) = b + 1$.

From the graph in Figure 3.22 (using the bipartite graph representation), we see that $n(2, b, 1) \geq b + 1$. To show we cannot put more nodes in the same graph, consider an initial graph consisting of the top node and top two buses. A node which is on the left bus must share a bus with all nodes on the right bus. But each of those nodes must share a bus with all nodes on the left bus. As a result, all second-tier nodes must share a common bus, so there can be only b. Adding the top node gives $b + 1$ total nodes.

3. $n(2, b, 2) = b^2 + 1$.

The graph in Figure 3.23 shows $n(2, b, 2) \geq b^2 + 1$, for $b = 5$. The analogous construction for other b's is obvious. To show this is the maximum possible number, look at Figure 3.24, which shows the largest potential graph from a particular node. Call the nodes below a level 3/2 bus a group. Every left level 2 node must share a bus with some node in each right group, in order to reach every right level one node. The converse applies to each right level 2 node. Thus every bus must contain a representative from every group. As a result, there can be at most b groups. But then there are $1 + b + b(b-1) = b^2 + 1$ nodes.

In both cases 2 and 3 these graphs, which are maximal, are about half as large as the Moore bound.

For several values of r and b, the value of $n(r, b, 1)$ can be determined. For example, $n(3, 4, 1) = 8$, as shown in Figure 3.25. Several other examples are given in [19].

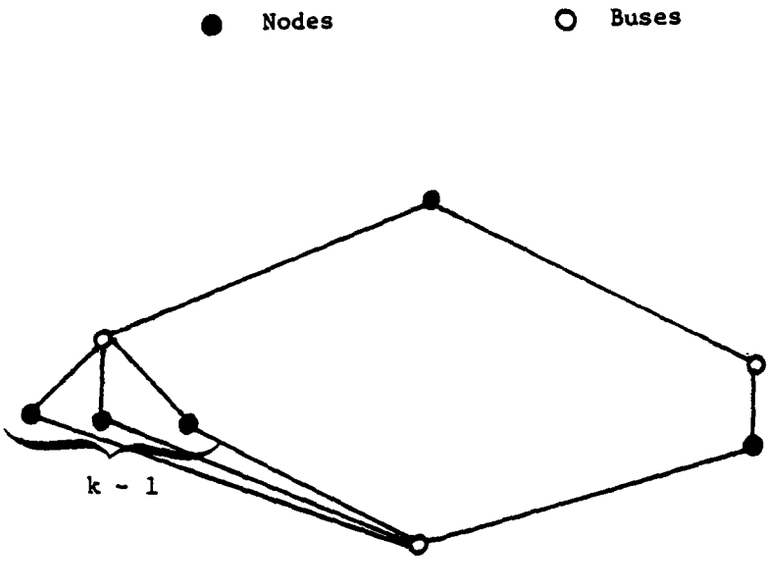


Figure 3.22. Graph Showing $n(2, b, 1) \geq b + 1$

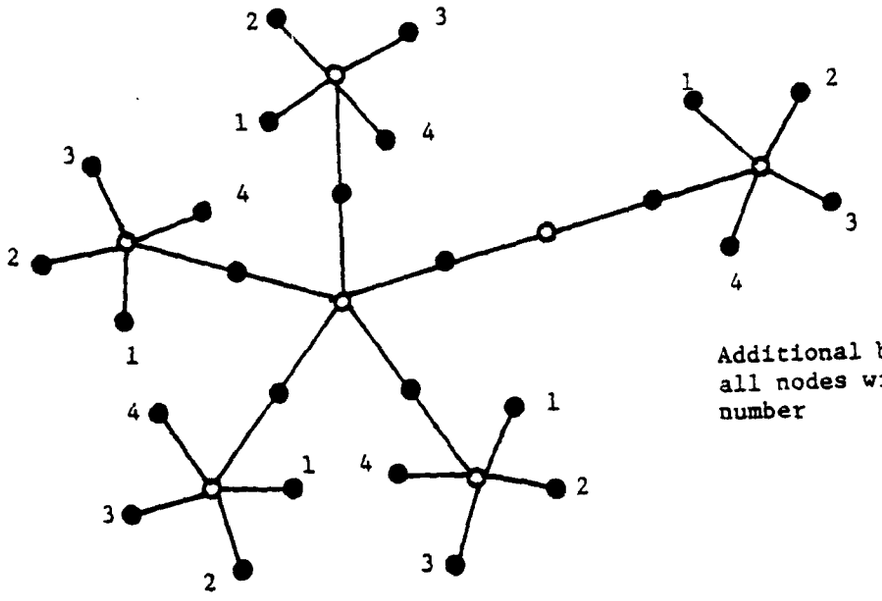


Figure 3.23. Graph Showing $n(2, b, 2) \geq b^2 + 1$

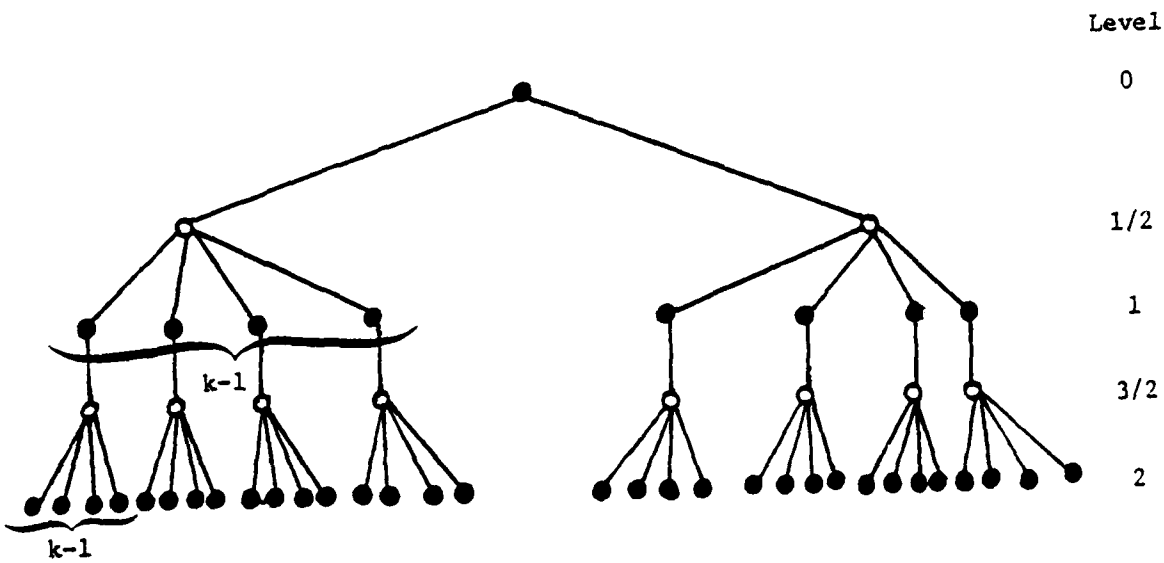
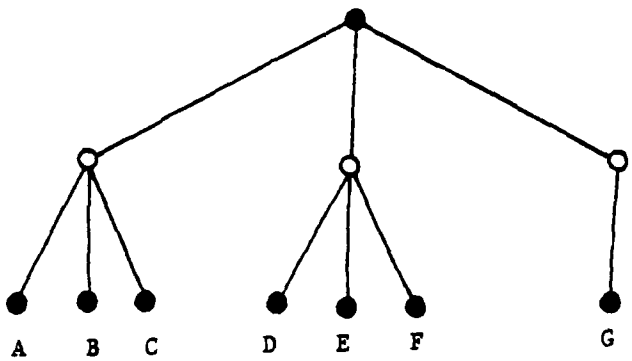


Figure 3.24. Graph Showing $n(2, b, 2) \leq b^2 + 1$



Additional buses connect ABEF; ABDG; CEFG; CD.

Figure 3.25. Graph Showing $n(3, 4, 1) = 8$

For larger networks, general construction methods must be developed. While there will not produce maximal graphs, they will produce reasonably large ones. The basic method is to use special cases of the graphs described in Section 3.3 which are bipartite. Certain star polygons, multi-tree structured networks, hingings, and chordal rings are bipartite. One type of node is designated the processors, and the other type is designated the buses. The diameter can then be calculated, and the graph evaluated. More details on these methods can be found in [19].

Figure 3.26 shows a hinging example. In this type of graph the processor nodes and bus nodes can have different degrees. Here $d = 3$, $b = 4$, and the diameter $k = 3$. There are 40 processor nodes.

Figure 3.27 shows how a chordal ring can represent a bus connection network. In this case the bus degree and processor degree must be equal. Here there are 24 processor nodes and a diameter of 2.

The graphs based on de Bruijn sequences can be generalized for use as bus connection networks. These networks will have a size of $(db/4)^k$, when d and b are both even. In addition, some of the graph products can be used with bus connection networks. Both of these topics are discussed in [19].

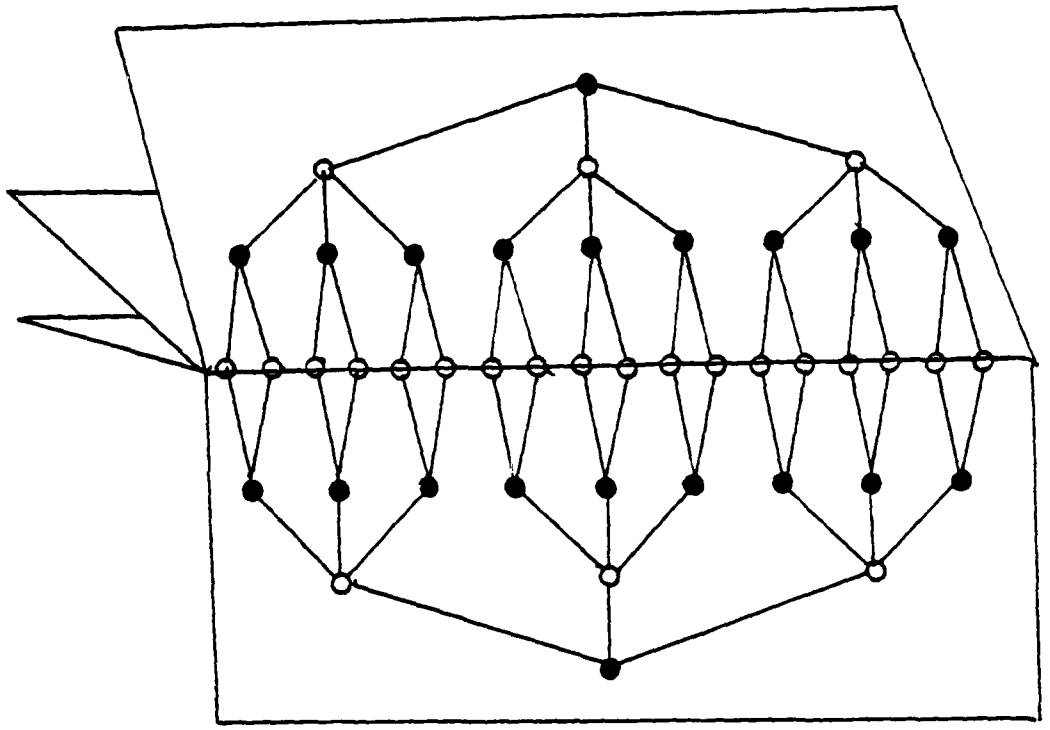


Figure 3.26. A Hinging With r = 3, b = 4, k = 3

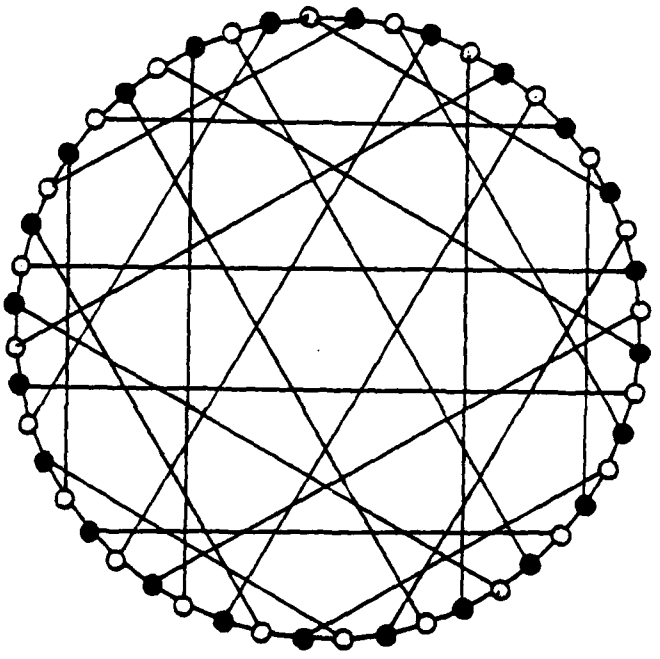


Figure 3.27. Chordal Ring With Degree 3, Diameter 2, 24 Processor Nodes

REFERENCES

[1] AKERS, S. (1965). On the Construction of (d,k) Graphs. IEEE Trans. Electron. Comput. Vol. EC-14, p. 448.

[2] ARDEN, B. and LEE, H. (1981). Analysis of Chordal Ring Network. IEEE Trans. Comput. Vol. C-30, pp. 291-295.

[3] ARDEN, B. and LEE, H. (1982). A Regular Network for Multicomputer Systems. IEEE. Trans. Comput. Vol. C-31, pp. 60-69.

[4] BAKER, K. and SU, Z. (1974). Sequencing with Due Dates and Early Start Times to Minimize Maximum Tardiness. Nav. Res. Log. Quart. Vol.21, pp. 171-176.

[5] BANNAI, E. and ITO, T. (1973). On Finite Moore Graphs. J. Fac. Sci. Univ. Tokyo Vol. 20, pp. 191-208.

[6] BANNAI, E. and ITO, T. (1981). Regular Graphs with Excess One. Discrete Math. Vol. 37, pp. 147-158.

[7] BASKETT, F., CHANDY, K.M., MUNTZ, R.R. and PALACIOS, F.G. (1975). Open, Closed and Mixed Networks of Queues with Different Classes of Customers. J. Assoc. Comput. Mach. Vol. 22, pp. 248-260.

[8] BOKHARI, S.H. (1981). A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System. IEEE Trans. Software Engr. Vol. SE-7, pp. 583-589.

[9] BOLLOBAS, B. (1978). Extremal Graph Theory. Academic Press, London.

[10] BOSE, R. and DOWLING, T. (1971). A Generalization of Moore Graphs of Diameter Two. J. Combinational Theory Vol. 11, pp. 213-226.

[11] BRUCK, R. and RYSER, H. (1949). The Nonexistence of Certain Finite Projective Planes. Canadian J. Math. Vol. 1, pp. 88-93.

[12] CHUNG, K.L. (1974). A Course in Probability Theory. 2nd Ed. Academic Press, New York.

[13] CINLAR, E. (1972). Superpostion of Point Processes. In Stochastic Point Processes: Statistical Analysis, Theory, and Applications. P.A.W. Lewis, Editor. Wiley, New York. pp. 549-606.

[14] COCKAYNE, E.J., RUSKEY, F. and THOMASON, A.G. (1979). An Algorithm for the Most Economic Link Addition in a Tree Communications Network. Inform. Proc. Letters Vol. 9, pp. 171-175.

[15] DAMERELL, R. (1973). On Moore Graphs. Proc. Camb. Phil. Soc. Vol.'74, pp. 227-236.

[16] deBRUIJN, D.G. (1946). A Combinatorial Problem. Nerderl. Akad. Wentensch. Proc. Ser. A49, pp. 758-764.

[17] DOTY, K.W., McENTIRE, P.L. and O'REILLY, J.G. (1981). Design Methodology Study for Airborne Distributed Data Processing Systems. Final Report, Contract No. N62269-80-C-0121, Systems Control, Inc.

[18] DOTY, K.W. (1982). Construction Methods for Asymptotically Large Graphs. Tech Memo. 5452-3, Systems Control Technology.

[19] DOTY, K.W. (1982). Dense Buss Connection Networks. Tech. Memo. 5452-2, Systems Control Technology.

[20] DOTY, K.W. (1982). Large Regular Interconnection Networks. (To be presented at the Third International Conference on Distributed Computing Systems, Hollywood, Florida.)

[21] ELSPAS, B. (1964). Topological Constraints on Interconnection-Limited Logic. Switching Theory Logic Design Vol. S-164, pp. 133-147.

[22] FINKEL, R.A. and SOLOMON, M.H. (1980). Processor Interconnection Strategies. IEEE Trans. Computers Vol. C-29, pp. 360-371.

[23] FISHER, R. and YATES, F. (1963). Statistical Tables for Biological, Agricultural and Medical Research. Hafner, New York.

[24] FLOYD, R.W. (1962). Algorithm 97: Shortest Path. Comm. ACM Vol. 5, p. 345.

[25] FRIEDMAN, H. (1966). A Design for (d,k) Graphs. IEEE Trans. Electron. Comput. Vol. EC-15, pp. 253-254.

[26] FUGLISTER, F.J. (1977). On Finite Moore Geometries. J. Combinatorial Theory Vol. 23, pp. 187-197.

[27] HALL, M. (1967). Combinatorial Theory. Blaisdell, Waltham, Mass.

[28] HANANI H. (1961). The Existence and Construction of Balanced Incomlete Block Designs. Ann. Math. Statist. Vol. 32, pp. 361-386.

[29] HARRISON, J.M. and LEMOINE, A.J. (1981). A Note on Networks of Infinite-Server Queues. J. Appl. Prob. Vol. 18, pp. 561-567.

[30] HOEL, P.G., PORT, S.C. and STONE, C.J. (1971). Introduction to Probability Theory. Houghton Mifflin, Boston.

[31] HOFFMAN, A. and SINGLETON, R. (1960). On Moore Graphs with Diameters 2 and 3. IBM J. Res. Develop. Vol. 4, pp. 497-504.

[32] HU, T.C. (1961). Parallel Sequencing and Assembly Line Problems. Operat. Res. Vol. 9, pp. 841-848.

[33] IMASE, M. and ITOH, M. (1981). Design to Minimize Diameter on Building-Block Network. IEEE Trans. Comput. Vol. C-30, pp. 439-442.

[34] KAUFMAN, M.T. (1973). Efficient Near-Optimal Scheduling of Multi-Processor Systems. Ph.D. Dissertation, Department of Computer Science, Stanford University.

[35] KEMENY, J.G. and SNELL, J.L. (1960). Finite Markov Chains. Van Nostrand, Princeton, N.J..

[36] KORN, I. (1967). On (d,k) Graphs. IEEE Trans. Electron. Comput. Vol. EC-16, p. 90.

[37] LELAND, W. et al. (1981). High Density Graphs for Processor Interconnection. Information Processing Letters Vol. 12, pp. 117-120.

[38] LENSTRA, J.K. and RINNOOY KAN, A.H.G. (1978). Complexity of Scheduling Under Precedence Constraints. Operat. Res. Vol. 26, pp. 22-35.

[39] McENTIRE, P.L. and LARSON, R.E. (1981). Optimal Resource Allocation in Sparse Networks. IFAC/81 Congress, Kyoto, Japan.

[40] MA, P.R., LEE, E.Y.S. and TSUCHIYA, M. (1982). A Task Allocation Model for Distributed Computing Systems. IEEE Trans. Comput. Vol. C-31, pp. 41-47.

[41] McMAHON, G. and FLORIAN, M. (1974). On Scheduling with Ready Times and Due Dates to Minimize Maximum Lateness. Report, Département d'Informatique, Université de Montréal.

[42] MICKUNAS, M.O. (1980). Using Projective Geometry to Design Bus Connection Networks. Proceedings of the Workshop on Interconnection Networks for Parallel and Distributed Processing. West Lafayette, Indiana, pp. 47-55.

[43] SCHULUMBERGER, M.A. (1974). de Bruijn Communication Networks. Ph.D. Dissertation, Department of Computer Science, Stanford University.

[44] STORWICK, R. (1970). Improved Connection Techniques for (d,k) Graphs. IEEE Trans. Comput. Vol. C-19, pp. 1214-1216.

[45] VEBLEN, O. and BUSSEY, W. (1906). Finite Projective Geometries. Trans. Amer. Math. Soc. Vol. 7, pp. 241-259.

[46] WITTIE, L.D. (1981). Communication Structures for Large Networks of Microcomputers. IEEE Trans. Comput. Vol. C-30, pp. 264-272.

[47] BALL, M.O. (1979). Computing Network Reliability. Operat. Res. Vol. 27, pp. 823-838.

[48] TORNG, H.C. and WILHELM, N.C. (1977). The Optimal Interconnection of Circuit Modules in Microprocessor and Digital System Design. IEEE Trans. Comput. Vol. C-26, pp. 450-457.

APPENDIX A

SPATIAL DYNAMIC PROGRAMMING COMPUTER CODE FOR SOFTWARE ALLOCATION

```
PROGRAM SDP2(INPUT,OUTPUT,INFILE,OUTFILE);

{------ Declarations ------}

%INCLUDE 'SDP2.DEC'

{-------------------------------------------------------------------}
PROCEDURE READDATA( VAR NROOT : PTR_TO_NODE_REC;
          VAR LROOT : PTR_TO_LINK_REC;
          VAR CROOT : PTR_TO_COM_REC); EXTERN;
{-------------------------------------------------------------------}
PROCEDURE FIND_COST( VAR NROOT : PTR_TO_NODE_REC;
          VAR LROOT : PTR_TO_LINK_REC;
          VAR CROOT : PTR_TO_COM_REC;
          VAR J_COST : COST_REC;
          VAR CUT : INTEGER); EXTERN;
{-------------------------------------------------------------------}
PROCEDURE WRITEDATA( VAR NROOT : PTR_TO_NODE_REC;
          VAR LROOT : PTR_TO_LINK_REC;
          VAR CROOT : PTR_TO_COM_REC); EXTERN;
{-------------------------------------------------------------------}
BEGIN

  PROMPT := ' In what file is input data ?            ';
  WRITE( PROMPT:30 );
  READLN( PROMPT ); WRITELN;
  OPEN(INFILE,FILE_NAME := PROMPT, HISTORY := OLD );
  RESET(INFILE);

  PROMPT := ' To what file should output be written ? ';
  WRITE( PROMPT:41 );
  READLN( PROMPT ); WRITELN;
  OPEN(OUTFILE,FILE_NAME := PROMPT, HISTORY := NEW );
  REWRITE(OUTFILE);

  NROOT := NIL;
  LROOT := NIL;
  CROOT := NIL;
  READDATA(NROOT, LROOT, CROOT);
  J_COST.LORD := NIL;
  J_COST.DIM := 0;
  NEW(J_COST.RO);
  J_COST.RO^.R[0] := 0.0;
  CUT := 0;
  FIND_COST( NROOT, LROOT, CROOT, J_COST, CUT );
  WRITEDATA(NROOT, LROOT, CROOT);
END.
```

```
MODULE READDATA(INPUT,OUTPUT,INFILE,OUTFILE);

{------ Declarations ------}

%INCLUDE 'SDP2.DEC'

{------ Procedures ------}

PROCEDURE READWORD( VAR WORD : PACKED ARRAY [SUB2] OF CHAR;
          J : INTEGER);
CONST
    BLANK = ' ';
VAR
    CHR : CHAR;
    I : INTEGER;
    ENCOUNTER : BOOLEAN;
BEGIN
  ENCOUNTER := FALSE;
  I := 0;
  REPEAT
    READ(INFILE,CHR);
    IF(CHR <> BLANK) THEN ENCOUNTER := TRUE;
    IF((ENCOUNTER) AND (CHR <> BLANK)) THEN
      BEGIN
        I := I + 1;
        IF (I <= J) THEN WORD[I] := CHR;
      END;
  UNTIL ((EOLN(INFILE)) OR ((ENCOUNTER) AND (CHR = BLANK)));
  I := I + 1;
  WHILE (I <= J) DO
    BEGIN
      WORD[I] := BLANK;
      I := I + 1;
    END;
END;
{-------------------------------------------------------------}
PROCEDURE FIND_NODE( VAR NROOT : PTR_TO_NODE_REC;
          VAR CUR_NODE : PTR_TO_NODE_REC;
          VAR NODE : INTEGER); EXTERN;
{-------------------------------------------------------------}
PROCEDURE READ_NODES( VAR NROOT : PTR_TO_NODE_REC;
          VAR J : INTEGER;
          VAR N : INTEGER);
VAR
  C : PACKED ARRAY [1..1] OF CHAR;
BEGIN
  IF( J > 0 ) THEN
  BEGIN
    N := N + 1;
    NEW(NROOT);
    NROOT^.NEXT := NIL;
    WITH NROOT^ DO
    BEGIN
      READ(INFILE, NAME );
      IF( NAME <> N ) THEN
      BEGIN
        WRITELN(' Error - READ_NODES - faulty node order in input file');
        WRITELN(' incorrect node number = ':25,NAME:3,
          ' should be = ':13,N:3);
      END;
      READWORD(C,1);
```

```
      CASE C[1] OF
        'P','p' : BEGIN
                    KIND := PRO;
                    READLN(INFILE,MEM_CAP,SPEED);
                    TIME_USE := -1.0;
                    MEM_USE := -1;
                  END;
        'M','m' : BEGIN
                    KIND := MODUL;
                    READLN(INFILE,MEM_REQ,INSTR,TIME_CONSTRAINT);
                  END;
        OTHERWISE
        BEGIN
          READLN(INFILE);
          WRITELN(' Error - READ_NODES -':21,
            ' incorrect kind of node in input file');
          WRITELN(' node # = ':10,NAME:3,',  1st character = ':19,C:1);
          WRITELN(' Possible error due to incorrect number of nodes':48,
            ' in input file':14);
        END;
      END;
    END;
    J := J - 1;
    READ_NODES( NROOT^.NEXT, J, N);
  END;
END;
{-------------------------------------------------------------------------}
PROCEDURE READ_LINKS( VAR LROOT : PTR_TO_LINK_REC;
          VAR NROOT : PTR_TO_NODE_REC;
          VAR J : INTEGER);
VAR
  NODET : INTEGER;
  N1, N2 : PTR_TO_NODE_REC;
BEGIN
  IF( J > 0 ) THEN
  BEGIN
    IF( EOF(INFILE) ) THEN
    BEGIN
      WRITELN(' Error - READ_LINKS - end of input file encountered before':58);
      WRITELN('   end of anticipated link list');
      WRITELN(' Possible error due to incorrect number of links':48,
        ' in input file':14);
    END;
    N1 := NIL;
    N2 := NIL;
    NEW(LROOT);
    LROOT^.NEXT := NIL;
    WITH LROOT^ DO
    BEGIN
      READLN(INFILE, NODE1, NODE2 );
      FIND_NODE( NROOT, N1, NODE1 );
      FIND_NODE( NROOT, N2, NODE2 );
      IF( N1 = NIL ) THEN
        WRITELN(' Error - READ_LINKS - node ':27,NODE1:3,' does not exist':15);
      IF( N2 = NIL ) THEN
        WRITELN(' Error - READ_LINKS - node ':27,NODE2:3,' does not exist' 15);
      IF(( N1 <> NIL ) AND ( N2 <> NIL )) THEN
      BEGIN
        IF( N1^.KIND = N2^.KIND ) THEN
        BEGIN
          WRITELN(' Error - READ_LINKS - Input file indicates that');
```

```
            CASE N1^.KIND OF
              PRO    : WRITELN('   processor ':13,N1^.NAME:3,' is to join ':12,
                       ' processor ':11,N2^.NAME:3);
              MODUL  : WRITELN('   memory ':10,N1^.NAME:3,' is to join ':12,
                       ' memory ':8,N2^.NAME:3);
              OTHERWISE
                WRITELN(' Error - READ_LINKS - CASE on KINDs');
            END;
          END;
        END;
        IF( NODE1 > NODE2 ) THEN
        BEGIN
          NODET := NODE1;
          NODE1 := NODE2;
          NODE2 := NODET;
        END;
        CONNECT := -1000000;
        DIM := -1;  { keep }
      END;
      J := J - 1;
      READ_LINKS( LROOT^.NEXT, NROOT, J);
    END;
  END;
  {-------------------------------------------------------------------------}
  PROCEDURE READ_COM_LINKS( VAR CROOT : PTR_TO_COM_REC;
            VAR NROOT : PTR_TO_NODE_REC;
            VAR J : INTEGER);
  VAR
    N1, N2 : PTR_TO_NODE_REC;
  BEGIN
    IF( J > 0 ) THEN
    BEGIN
      IF( EOF(INFILE) ) THEN
      BEGIN
        WRITELN(' Error - READ_COM_LINKS - end of input file encountered before');
        WRITELN('   end of anticipated communication link list');
      END;
      NEW(CROOT);
      WITH CROOT^ DO
      BEGIN
        READLN(INFILE,MOD1,MOD2,DELAY);
        IF( MOD1 = MOD2 ) THEN
        BEGIN
          WRITELN(' Error - READ_COM_LINKS - input file indicates that');
          WRITELN('   module ':10,MOD1:3,' must communicate to itself':27);
        END;
        PMOD1 := NIL;
        PMOD2 := NIL;
        N1 := NIL;
        N2 := NIL;
        CONSIDER := FALSE;
        NEXT := NIL;
        FIND_NODE( NROOT, N1, MOD1 );
        FIND_NODE( NROOT, N2, MOD2 );
        IF( N1 = NIL ) THEN
          WRITELN(' Error - READ_COM_LINKS - node ':31,MOD1:3,
            ' does not exist in node list':28)
        ELSE
          IF( N1^.KIND <> MODUL ) THEN
            WRITELN(' Error - READ_COM_LINKS - node ':31,MOD1:3,
              ' is not a module':16);
```

```
        IF( N2 = NIL ) THEN
          WRITELN(' Error - READ_COM_LINKS - node ':31,MOD2:3,
            ' does not exist in node list':28)
        ELSE
          IF( N2^.KIND <> MODUL ) THEN
            WRITELN(' Error - READ_COM_LINKS - node ':31,MOD2:3,
              ' is not a module':16);
      END;
      J := J - 1;
      READ_COM_LINKS( CROOT^.NEXT, NROOT, J );
    END
    ELSE
    BEGIN
      IF( NOT EOF(INFILE) ) THEN
      BEGIN
        WRITELN(' Error - READ_COM_LINKS -':25,
          ' last communication link(s) were not read in input file');
        WRITELN(' Possible error due to incorrect number of':42,
          ' communication links in input file':34);
      END;
    END;
  END;
  {-------------------------------------------------------------------------}
  PROCEDURE READDATA( VAR NROOT : PTR_TO_NODE_REC;
            VAR LROOT : PTR_TO_LINK_REC;
            VAR CROOT : PTR_TO_COM_REC);
  VAR
    J,N : INTEGER;
  BEGIN
    READLN(INFILE,J);
    N := 0;
    READ_NODES( NROOT, J, N );
    READ(INFILE,J);
    IF( NOT EOLN(INFILE) ) THEN
    BEGIN
      WRITELN(' Error - READDATA - blank or extra data following number':56,
        ' of links in input file':23);
      WRITELN(' Possible error due to incorrect number of nodes':48,
        ' in input file':14);
    END;
    READLN(INFILE);
    READ_LINKS( LROOT, NROOT, J );
    IF( EOF(INFILE) ) THEN
      WRITELN(' Inform - READDATA - input file contains no':43,
        ' communication link constraints':31)
    ELSE
    BEGIN
      READ(INFILE,J);
      IF( NOT EOLN(INFILE) ) THEN
      BEGIN
        WRITELN(' Error - READDATA - blank or extra data following number':56,
          ' of communication links in input file':37);
        WRITELN(' Possible error due to incorrect number of links':48,
          ' in input file':14);
      END;
      READLN(INFILE);
      READ_COM_LINKS( CROOT, NROOT, J );
    END;
  END;
  {-------------------------------------------------------------------------}
  END.
```

```
MODULE FIND(INPUT,OUTPUT,INFILE,OUTFILE);

{------ Declarations ------}

%INCLUDE 'SDP2.DEC'

{-------------------------------------------------------------------------}
PROCEDURE FIND_NODE( VAR NROOT : PTR_TO_NODE_REC;
          VAR CUR_NODE : PTR_TO_NODE_REC;
          VAR NODE : INTEGER);
BEGIN
  IF( NROOT <> NIL ) THEN
  BEGIN
    IF( NROOT^.NAME = NODE ) THEN CUR_NODE := NROOT
    ELSE FIND_NODE( NROOT^.NEXT, CUR_NODE, NODE );
  END;
END;
{-------------------------------------------------------------------------}
PROCEDURE FIND_EXTERNALS( VAR LROOT : PTR_TO_LINK_REC;
          VAR LORD : PTR_TO_LINK_ORD;
          VAR DIM : INTEGER;
          VAR CUT : INTEGER);
BEGIN
  IF( LROOT <> NIL ) THEN
  BEGIN
    IF(( LROOT^.NODE1 <= CUT ) AND ( CUT < LROOT^.NODE2 )) THEN
    BEGIN
      NEW(LORD);
      LORD^.LINK := LROOT;
      LORD^.NODE := NIL;  { not currently needed }
      LORD^.NEXT := NIL;
      DIM := DIM + 1;
      FIND_EXTERNALS( LROOT^.NEXT, LORD^.NEXT, DIM, CUT);
    END
    ELSE FIND_EXTERNALS( LROOT^.NEXT, LORD, DIM, CUT );
  END;
END;
{-------------------------------------------------------------------------}
PROCEDURE FIND_INTERNALS( VAR LROOT : PTR_TO_LINK_REC;
          VAR LORD : PTR_TO_LINK_ORD;
          VAR DIM : INTEGER;
          VAR CUT : INTEGER;
          VAR DEG : INTEGER);
BEGIN
  IF( LROOT <> NIL ) THEN
  BEGIN
    IF( LROOT^.NODE2 = CUT ) THEN
    BEGIN
      NEW(LORD);
      LORD^.LINK := LROOT;
      LORD^.NODE := NIL;  { not currently needed }
      LORD^.NEXT := NIL;
      IF( LORD^.LINK^.DIM = -1 ) THEN
      BEGIN
        LORD^.LINK^.DIM := DEG;
        CASE DEG OF
          0 : NEW(LORD^.LINK^.I0);
          1 : NEW(LORD^.LINK^.I1);
          2 : NEW(LORD^.LINK^.I2);
          3 : NEW(LORD^.LINK^.I3);
          4 : NEW(LORD^.LINK^.I4);
```

```
            5 : NEW(LORD^.LINK^.I5);
            6 : NEW(LORD^.LINK^.I6);
            7 : NEW(LORD^.LINK^.I7);
            8 : NEW(LORD^.LINK^.I8);
            9 : NEW(LORD^.LINK^.I9);
            OTHERWISE
            BEGIN
              WRITELN(' Error - FIND_INTERNALS - DEG is out of bounds');
              WRITELN(' CUT = ':7,CUT:3,',    DEG = ':10,DEG:3);
            END;
          END;
        END
        ELSE
        BEGIN
          WRITELN(' Error - FIND_INTERNALS - DIM is already defined');
          WRITELN(' CUT = ':7,CUT:3,
            ',  LORD^.LINK^.DIM = ':21, LORD^.LINK^.DIM:3);
        END;
        DIM := DIM + 1;
        FIND_INTERNALS( LROOT^.NEXT, LORD^.NEXT, DIM, CUT, DEG);
      END
      ELSE FIND_INTERNALS( LROOT^.NEXT, LORD, DIM, CUT, DEG);
    END;
  END;
  {-------------------------------------------------------------------------}
  PROCEDURE FIND_CUT_LINK( VAR NROOT : PTR_TO_NODE_REC;
            VAR LROOT : PTR_TO_LINK_REC;
            VAR LORD : PTR_TO_LINK_ORD;
            VAR DIM : INTEGER;
            VAR CUT : INTEGER);
  BEGIN
    IF( LROOT <> NIL ) THEN
    BEGIN
      IF(( CUT = LROOT^.NODE1 ) OR ( CUT = LROOT^.NODE2 )) THEN
      BEGIN
        NEW(LORD);
        LORD^.LINK := LROOT;
        LORD^.NODE := NIL;
        IF( CUT = LROOT^.NODE1 ) THEN
          FIND_NODE( NROOT, LORD^.NODE, LROOT^.NODE2)
        ELSE
          FIND_NODE( NROOT, LORD^.NODE, LROOT^.NODE1);
        IF( LORD^.NODE = NIL ) THEN
          WRITELN(' Error - FIND_CUT_LINK - LORD^.NODE = NIL');
        LORD^.NEXT := NIL;
        DIM := DIM + 1;
        FIND_CUT_LINK( NROOT, LROOT^.NEXT, LORD^.NEXT, DIM, CUT);
      END
      ELSE FIND_CUT_LINK( NROOT, LROOT^.NEXT, LORD, DIM, CUT );
    END;
  END;
  {-------------------------------------------------------------------------}
  PROCEDURE SIFT_COM_LINKS( VAR COM : PTR_TO_COM_REC;
            VAR CROOT : PTR_TO_COM_REC;
            VAR LORD : PTR_TO_LINK_ORD);
  BEGIN
    IF( LORD <> NIL ) THEN
    BEGIN
      IF(( CROOT^.MOD1 = LORD^.LINK^.NODE1) OR
      ( CROOT^.MOD1 = LORD^.LINK^.NODE2)) THEN
      BEGIN
```

```
        CROOT^.PMOD1 := LORD^.LINK;
        IF( COM = NIL ) THEN COM := CROOT;
      END;
      IF(( CROOT^.MOD2 = LORD^.LINK^.NODE1) OR
      ( CROOT^.MOD2 = LORD^.LINK^.NODE2)) THEN
      BEGIN
        CROOT^.PMOD2 := LORD^.LINK;
        IF( COM = NIL ) THEN COM := CROOT;
      END;
      SIFT_COM_LINKS( COM, CROOT, LORD^.NEXT );
    END;
  END;
  {-------------------------------------------------------------------------}
  PROCEDURE FIND_COM_LINKS( VAR CROOT : PTR_TO_COM_REC;
            VAR CORD : PTR_TO_COM_ORD;
            VAR LORD : PTR_TO_LINK_ORD;
            VAR CUT : INTEGER);
  BEGIN
    IF( CROOT <> NIL ) THEN
    BEGIN
      NEW( CORD );
      CORD^.COM := NIL;
      CORD^.NEXT := NIL;
      CROOT^.PMOD1 := NIL;
      CROOT^.PMOD2 := NIL;
      SIFT_COM_LINKS( CORD^.COM, CROOT, LORD );
      IF( CORD^.COM = NIL ) THEN
      BEGIN
        DISPOSE( CORD );
        FIND_COM_LINKS( CROOT^.NEXT, CORD, LORD, CUT);
      END
      ELSE
      BEGIN
        FIND_COM_LINKS( CROOT^.NEXT, CORD^.NEXT, LORD, CUT);
        IF( CROOT^.CONSIDER = TRUE ) THEN
        ` WRITELN(' Error - FIND_COM_LINKS - modules ':34,CROOT^.MOD1:3,
            ' and ':5,CROOT^.MOD2:3,' have CONSIDER = true':21,
            ' at CUT = ':10,CUT:3);
        CROOT^.CONSIDER := FALSE;
      END;
    END;
  END;
  {-------------------------------------------------------------------------}
  END.
```

```
MODULE COST(INPUT,OUTPUT,INFILE,OUTFILE);

{------ Declarations ------}

%INCLUDE 'SDP2.DEC'

{-------------------------------------------------------------------------}
PROCEDURE FIND_NODE( VAR NROOT : PTR_TO_NODE_REC;
          VAR CUR_NODE : PTR_TO_NODE_REC;
          VAR NODE : INTEGER); EXTERN;
{-------------------------------------------------------------------------}
PROCEDURE FIND_INTERNALS( VAR LROOT : PTR_TO_LINK_REC;
          VAR LORD : PTR_TO_LINK_ORD;
          VAR DIM : INTEGER;
          VAR CUT : INTEGER;
          VAR DEQ : INTEGER); EXTERN;
{-------------------------------------------------------------------------}
PROCEDURE FIND_EXTERNALS( VAR LROOT : PTR_TO_LINK_REC;
          VAR LORD : PTR_TO_LINK_ORD;
          VAR DIM : INTEGER;
          VAR CUT : INTEGER); EXTERN;
{-------------------------------------------------------------------------}
PROCEDURE FIND_CUT_LINK( VAR NROOT : PTR_TO_NODE_REC;
          VAR LROOT : PTR_TO_LINK_REC;
          VAR LORD : PTR_TO_LINK_ORD;
          VAR DIM : INTEGER;
          VAR CUT : INTEGER); EXTERN;
{-------------------------------------------------------------------------}
PROCEDURE COMPUTE_K_COST( VAR K_COST : COST_REC;
          VAR CUR_NODE : PTR_TO_NODE_REC;
          VAR CUT : INTEGER;
          VAR CORD : PTR_TO_COM_ORD); EXTERN;
{-------------------------------------------------------------------------}
PROCEDURE RETRIEVE_VAR( VAR LORD : PTR_TO_LINK_ORD;
          VAR J : INTEGER); EXTERN;
{-------------------------------------------------------------------------}
PROCEDURE COMPUTE_J_COST( VAR LROOT : PTR_TO_LINK_REC;
          VAR J_COST : COST_REC;
          VAR J_PREV_COST : COST_REC;
          VAR K_COST : COST_REC;
          VAR INTERNALS : VAR_ORD;
          VAR CUR_NODE : PTR_TO_NODE_REC); EXTERN;
{-------------------------------------------------------------------------}
PROCEDURE LOAD_CONNECT( VAR LORD : PTR_TO_LINK_ORD;
          VAR CUT : INTEGER;
          VAR J : INTEGER);
BEGIN
  IF( LORD <> NIL ) THEN
  BEGIN
    CASE LORD^.LINK^.DIM OF
      0 : LORD^.LINK^.CONNECT := LORD^.LINK^.I0^.I[J];
      1 : LORD^.LINK^.CONNECT := LORD^.LINK^.I1^.I[J];
      2 : LORD^.LINK^.CONNECT := LORD^.LINK^.I2^.I[J];
      3 : LORD^.LINK^.CONNECT := LORD^.LINK^.I3^.I[J];
      4 : LORD^.LINK^.CONNECT := LORD^.LINK^.I4^.I[J];
      5 : LORD^.LINK^.CONNECT := LORD^.LINK^.I5^.I[J];
      6 : LORD^.LINK^.CONNECT := LORD^.LINK^.I6^.I[J];
      7 : LORD^.LINK^.CONNECT := LORD^.LINK^.I7^.I[J];
      8 : LORD^.LINK^.CONNECT := LORD^.LINK^.I8^.I[J];
      9 : LORD^.LINK^.CONNECT := LORD^.LINK^.I9^.I[J];
      OTHERWISE
```

```
      BEGIN
        WRITELN(' Error - LOAD_CONNECT - LORD^.LINK^.DIM out of range');
        WRITELN(' CUT = ':7,CUT:3,
          ',    LORD^.LINK^.DIM = ':22, LORD^.LINK^.DIM:3);
      END;
    END;
    LOAD_CONNECT( LORD^.NEXT, CUT, J);
  END;
END;
{-------------------------------------------------------------------------}
PROCEDURE SHOW_LINKS( VAR INTERNALS : VAR_ORD;
          VAR J_COST : COST_REC;
          VAR CUT : INTEGER);
VAR
  J : INTEGER;
BEGIN
  J := 0;
  RETRIEVE_VAR( J_COST.LORD, J);
  LOAD_CONNECT( INTERNALS.LORD, CUT, J);
END;
{-------------------------------------------------------------------------}
PROCEDURE DOWN_K_COST( VAR LORD : PTR_TO_LINK_ORD;
          J : INTEGER;
          VAR CORD : PTR_TO_COM_ORD;
          VAR INSTRUCT, REQUIRE : INTEGER;
          VAR PENALTY : REAL;
          VAR CUT : INTEGER;
          VAR OK : BOOLEAN;
          VAR CUR_NODE : PTR_TO_NODE_REC;
          DO_CHECK_COM : BOOLEAN); EXTERN;
{-------------------------------------------------------------------------}
PROCEDURE SHOW_COST( VAR CUR_NODE : PTR_TO_NODE_REC;
          VAR CORD : PTR_TO_COM_ORD;
          VAR K_COST : COST_REC;
          VAR CUT : INTEGER);
VAR
  J, INSTRUCT, REQUIRE : INTEGER;
  PENALTY : REAL;
  OK : BOOLEAN;
BEGIN
  IF( CUR_NODE^.KIND = PRO ) THEN
  BEGIN
    J := 0;
    INSTRUCT := 0; { not used }
    REQUIRE := 0;
    PENALTY := 0.0; { not used }
    OK := TRUE; { not used }
    RETRIEVE_VAR( K_COST.LORD, J);
    DOWN_K_COST( K_COST.LORD, J, CORD, INSTRUCT, REQUIRE,
      PENALTY, CUT, OK, CUR_NODE, FALSE );
    CASE K_COST.DIM OF
      0 : CUR_NODE^.TIME_USE := K_COST.R0^.R[J];
      1 : CUR_NODE^.TIME_USE := K_COST.R1^.R[J];
      2 : CUR_NODE^.TIME_USE := K_COST.R2^.R[J];
      3 : CUR_NODE^.TIME_USE := K_COST.R3^.R[J];
      4 : CUR_NODE^.TIME_USE := K_COST.R4^.R[J];
      5 : CUR_NODE^.TIME_USE := K_COST.R5^.R[J];
      6 : CUR_NODE^.TIME_USE := K_COST.R6^.R[J];
      7 : CUR_NODE^.TIME_USE := K_COST.R7^.R[J];
      8 : CUR_NODE^.TIME_USE := K_COST.R8^.R[J];
      9 : CUR_NODE^.TIME_USE := K_COST.R9^.R[J];
```

```
        OTHERWISE
        BEGIN
          WRITELN(' Error - SHOW_COST - K_COST.DIM out of range');
          WRITELN(' CUT = ':7,CUT:3,',    K_COST.DIM = ':17);
        END;
      END;
      CUR_NODE^.MEM_USE := REQUIRE;
    END;
  END;
  {-------------------------------------------------------------------------}
  PROCEDURE FIND_COM_LINKS( VAR CROOT: PTR_TO_COM_REC;
            VAR CORD : PTR_TO_COM_ORD;
            VAR LORD : PTR_TO_LINK_ORD;
            VAR CUT : INTEGER); EXTERN;
  {-------------------------------------------------------------------------}
  PROCEDURE REORDER_LORD( VAR LORD : PTR_TO_LINK_ORD );
  VAR TOP, MID, TEMP : PTR_TO_LINK_ORD;
  BEGIN
    NEW( TOP );
    TOP^.LINK := NIL;
    TOP^.NODE := NIL;
    TOP^.NEXT := LORD;
    LORD := NIL;
    WHILE( TOP^.NEXT <> NIL ) DO
    BEGIN
      TEMP := TOP;
      MID := TOP;
      WHILE( MID^.NEXT <> NIL ) DO
      BEGIN
        IF( TEMP^.NEXT^.NODE^.TIME_CONSTRAINT <=
        MID^.NEXT^.NODE^.TIME_CONSTRAINT ) THEN
          TEMP := MID;
        MID := MID^.NEXT;
      END;
      MID := TEMP^.NEXT;
      TEMP^.NEXT := MID^.NEXT;
      MID^.NEXT := LORD;
      LORD := MID;
    END;
    DISPOSE( TOP );
  END;
  {-------------------------------------------------------------------------}
  PROCEDURE FIND_COST( VAR NROOT : PTR_TO_NODE_REC;
            VAR LROOT : PTR_TO_LINK_REC;
            VAR CROOT : PTR_TO_COM_REC;
            VAR J_PREV_COST : COST_REC;
            VAR CUT : INTEGER);
  VAR
    J_COST, K_COST : COST_REC;
    CUR_NODE : PTR_TO_NODE_REC;
    INTERNALS : VAR_ORD;
    CORD : PTR_TO_COM_ORD;
  BEGIN
    CUT := CUT + 1;
    CUR_NODE := NIL;
    CORD := NIL;
    FIND_NODE( NROOT, CUR_NODE, CUT );
    IF( CUR_NODE = NIL ) THEN
    BEGIN
      WRITELN;
      WRITELN(' End of recursion. Preparing to back track. Cut = ':50,
```

```
            CUT:3);
        WRITELN;
        IF( J_PREV_COST.RO^.R[0] > 0.99*INFINITY ) THEN
        BEGIN
          WRITELN(OUTFILE,' Maximum cost is :  infinity !');
          WRITELN(OUTFILE,
            ' Probable cause due to non-connection of module to processor');
        END
        ELSE
          WRITELN(OUTFILE,' Maximum cost is : ',J_PREV_COST.RO^.R[0]:10:2);
      END
      ELSE
      BEGIN
        INTERNALS.DIM := 0;
        INTERNALS.LORD := NIL;
        J_COST.LORD := NIL;
        J_COST.DIM := 0;
        K_COST.LORD := NIL;
        K_COST.DIM := 0;
        FIND_EXTERNALS( LROOT, J_COST.LORD, J_COST.DIM, CUT );
        FIND_INTERNALS( LROOT, INTERNALS.LORD, INTERNALS.DIM, CUT, J_COST.DIM );
        FIND_CUT_LINK( NROOT, LROOT, K_COST.LORD, K_COST.DIM, CUT );
        CASE CUR_NODE^.KIND OF
          PRO    : BEGIN
                     REORDER_LORD( K_COST.LORD ); {must follow FIND_CUT_LINK}
{                    WRITELN(OUTFILE);}
                     FIND_COM_LINKS( CROOT, CORD, K_COST.LORD, CUT);
                     COMPUTE_K_COST( K_COST, CUR_NODE , CUT, CORD);
                     COMPUTE_J_COST( LROOT, J_COST, J_PREV_COST, K_COST,
                       INTERNALS, CUR_NODE );
                   END;
          MODUL  : BEGIN
                     IF( K_COST.DIM = 0 ) THEN
                     BEGIN
                       WRITELN(' Inform - FIND_COST - no link to current module');
                       WRITELN(' CUT = ',7,CUT:3);
                     END;
                     COMPUTE_J_COST( LROOT, J_COST, J_PREV_COST, K_COST,
                       INTERNALS, CUR_NODE );
                   END;
          OTHERWISE
          BEGIN
            WRITELN(' Error - FIND_COST - CUR_NODE^.KIND is invalid');
            WRITELN(' CUT = ',7,CUT:3,
              ',    CUR_NODE^.KIND = ',21,CUR_NODE^.KIND:5);
          END;
        END;
        FIND_COST( NROOT, LROOT, CROOT, J_COST, CUT );
        SHOW_LINKS( INTERNALS, J_COST, CUT );
        SHOW_COST( CUR_NODE, CORD, K_COST, CUT );
      END;
      CUT := CUT - 1;
    END;
    {-------------------------------------------------------------------------}
    END.
```

```
MODULE KCOST(INPUT,OUTPUT,INFILE,OUTFILE);

{------ Declarations ------}

%INCLUDE 'SDP2.DEC'

{-------------------------------------------------------------------------}
PROCEDURE CHECK_COM_LINKS( VAR NAME : INTEGER;
          VAR CORD : PTR_TO_COM_ORD;
          VAR PENALTY : REAL;
          VAR CUT : INTEGER);
BEGIN
  IF( CORD <> NIL ) THEN
  BEGIN
    WITH CORD^.COM^ DO
    BEGIN
      IF(( MOD1 = NAME) OR (MOD2 = NAME)) THEN
      BEGIN
        IF(( PMOD1 <> NIL) AND (PMOD2 <> NIL)) THEN
        BEGIN
          IF( CONSIDER = FALSE ) THEN
            CONSIDER := TRUE
          ELSE
          BEGIN
            CONSIDER := FALSE;
            IF( PMOD1^.CONNECT + PMOD2^.CONNECT = 1 ) THEN
              PENALTY := PENALTY + DELAY;
          END;
        END
        ELSE
        BEGIN
          IF(( PMOD1 = NIL) AND (PMOD2 = NIL)) THEN
          BEGIN
            WRITELN(' Error - CHECK_COM_LINKS - the SIFT_COM_LINKS':45,
              ' routine did not get':20);
            WRITELN(' pointers for either of the modules ':36,MOD1:3,
              ' and ':5,MOD2:3,' at cut = ':10,CUT:3);
          END
          ELSE
          BEGIN
            IF( PMOD1 <> NIL ) THEN IF( PMOD1^.CONNECT = 1 ) THEN
              PENALTY := PENALTY + DELAY;
            IF( PMOD2 <> NIL ) THEN IF( PMOD2^.CONNECT = 1 ) THEN
              PENALTY := PENALTY + DELAY;
          END;
        END;
      END;
    END;
    CHECK_COM_LINKS( NAME, CORD^.NEXT, PENALTY, CUT );
  END;
END;
{-------------------------------------------------------------------------}
PROCEDURE DOWN_K_COST( VAR LORD : PTR_TO_LINK_ORD;
          J : INTEGER;
          VAR CORD : PTR_TO_COM_ORD;
          VAR INSTRUCT, REQUIRE : INTEGER;
          VAR PENALTY : REAL;
          VAR CUT : INTEGER;
          VAR OK : BOOLEAN;
          VAR CUR_NODE : PTR_TO_NODE_REC;
          DO_CHECK_COM : BOOLEAN);
```

```
VAR
  K : INTEGER;
BEGIN
  IF( LORD <> NIL ) THEN
  BEGIN
    K := J MOD 2;
    J := J DIV 2;
    LORD^.LINK^.CONNECT := K;
    REQUIRE := REQUIRE + K*LORD^.NODE^.MEM_REQ;
    IF( DO_CHECK_COM = TRUE ) THEN
    BEGIN
      INSTRUCT := INSTRUCT + K*LORD^.NODE^.INSTR;
      CHECK_COM_LINKS( LORD^.NODE^.NAME, CORD, PENALTY, CUT);
      IF(( INSTRUCT/CUR_NODE^.SPEED + PENALTY >
      LORD^.NODE^.TIME_CONSTRAINT ) AND ( K = 1 )) THEN
        OK := FALSE;
    END;
    DOWN_K_COST( LORD^.NEXT, J, CORD, INSTRUCT, REQUIRE,
      PENALTY, CUT, OK, CUR_NODE, DO_CHECK_COM );
  END;
END;
{-------------------------------------------------------------------------}
PROCEDURE COMPUTE_K_COST( VAR K_COST : COST_REC;
          VAR CUR_NODE : PTR_TO_NODE_REC;
          VAR CUT : INTEGER;
          VAR CORD : PTR_TO_COM_ORD);
VAR
  LIMIT, INSTRUCT, REQUIRE, J : INTEGER;
  PENALTY : REAL;
  OK : BOOLEAN;
BEGIN
  IF(( 0 <= K_COST.DIM ) AND ( K_COST.DIM <= MAX_DIM )) THEN
  BEGIN
    CASE K_COST.DIM OF
      0 : BEGIN
            NEW(K_COST.R0);
            WRITELN(' Inform - COMPUTE_K_COST - no link to current processor');
            WRITELN(' CUT = ':7,CUT:3);
          END;
      1 : NEW(K_COST.R1);
      2 : NEW(K_COST.R2);
      3 : NEW(K_COST.R3);
      4 : NEW(K_COST.R4);
      5 : NEW(K_COST.R5);
      6 : NEW(K_COST.R6);
      7 : NEW(K_COST.R7);
      8 : NEW(K_COST.R8);
      9 : NEW(K_COST.R9);
      OTHERWISE
      BEGIN
        WRITELN(' Error - COMPUTE_K_COST - 1st group of CASEs');
      END;
    END;
    LIMIT := (2**K_COST.DIM) - 1;
    FOR J := 0 TO LIMIT DO
    BEGIN
      INSTRUCT := 0;
      REQUIRE := 0;
      PENALTY := 0.0;
      OK := TRUE;
      DOWN_K_COST( K_COST.LORD, J, CORD, INSTRUCT, REQUIRE,
```

```
      PENALTY, CUT, OK, CUR_NODE, TRUE ));
    IF(( REQUIRE <= CUR_NODE^.MEM_CAP ) AND ( OK = TRUE )) THEN
    BEGIN
      CASE K_COST.DIM OF
        0 : K_COST.R0^.R[J] := 0.0;  { ?? uncertain = infinity or 0? }
        1 : K_COST.R1^.R[J] := INSTRUCT/CUR_NODE^.SPEED + PENALTY;
        2 : K_COST.R2^.R[J] := INSTRUCT/CUR_NODE^.SPEED + PENALTY;
        3 : K_COST.R3^.R[J] := INSTRUCT/CUR_NODE^.SPEED + PENALTY;
        4 : K_COST.R4^.R[J] := INSTRUCT/CUR_NODE^.SPEED + PENALTY;
        5 : K_COST.R5^.R[J] := INSTRUCT/CUR_NODE^.SPEED + PENALTY;
        6 : K_COST.R6^.R[J] := INSTRUCT/CUR_NODE^.SPEED + PENALTY;
        7 : K_COST.R7^.R[J] := INSTRUCT/CUR_NODE^.SPEED + PENALTY;
        8 : K_COST.R8^.R[J] := INSTRUCT/CUR_NODE^.SPEED + PENALTY;
        9 : K_COST.R9^.R[J] := INSTRUCT/CUR_NODE^.SPEED + PENALTY;
        OTHERWISE
        BEGIN
          WRITELN(' Error - COMPUTE_K_COST - 2nd group of CASEs');
          WRITELN(' CUT = ':7,CUT:3,'    K_COST.DIM = ':17,K_COST.DIM:3);
        END;
      END;
    END
    ELSE
    BEGIN
      CASE K_COST.DIM OF
        0 : K_COST.R0^.R[J] := INFINITY;  { ?? uncertain = infinity or 0? }
        1 : K_COST.R1^.R[J] := INFINITY;
        2 : K_COST.R2^.R[J] := INFINITY;
        3 : K_COST.R3^.R[J] := INFINITY;
        4 : K_COST.R4^.R[J] := INFINITY;
        5 : K_COST.R5^.R[J] := INFINITY;
        6 : K_COST.R6^.R[J] := INFINITY;
        7 : K_COST.R7^.R[J] := INFINITY;
        8 : K_COST.R8^.R[J] := INFINITY;
        9 : K_COST.R9^.R[J] := INFINITY;
        OTHERWISE
        BEGIN
          WRITELN(' Error - COMPUTE_K_COST - 3rd group of CASEs');
          WRITELN(' CUT = ':7,CUT:3,'    K_COST.DIM = ':17,K_COST.DIM:3);
        END;
      END;
    END;
  END
  ELSE
  BEGIN
    WRITELN(' Error - COMPUTE_K_COST - K_COST.DIM out of range');
    WRITELN(' CUT = ':7,CUT:3,'    K_COST.DIM = ':17,K_COST.DIM:3);
  END;
END;
{-------------------------------------------------------------------------}
END.
```

```
MODULE JCOST(INPUT,OUTPUT,INFILE,OUTFILE);

{------ Declarations ------}

%INCLUDE 'SDP2.DEC'

{-------------------------------------------------------------------------}
PROCEDURE RECORD_VAR( VAR LORD : PTR_TO_LINK_ORD;
          J : INTEGER);
BEGIN
  IF( LORD <> NIL ) THEN
  BEGIN
    LORD^.LINK^.CONNECT := J MOD 2;
    J := J DIV 2;
    RECORD_VAR(LORD^.NEXT, J);
  END;
END;
{-------------------------------------------------------------------------}
PROCEDURE RETRIEVE_VAR( VAR LORD : PTR_TO_LINK_ORD;
          VAR J : INTEGER);
BEGIN
  IF( LORD <> NIL ) THEN
  BEGIN
    RETRIEVE_VAR( LORD^.NEXT, J );
    J := 2*J + LORD^.LINK^.CONNECT;
  END
  ELSE
    J := 0;
END;
{-------------------------------------------------------------------------}
FUNCTION MAX_COST( TEMP : REAL;
          VAR COST : COST_REC;
          VAR J : INTEGER;
          VAR CUT : INTEGER): REAL;
BEGIN
  MAX_COST := TEMP;
  CASE COST.DIM OF
    0 : IF( TEMP < COST.R0^.R[J] ) THEN MAX_COST := COST.R0^.R[J];
    1 : IF( TEMP < COST.R1^.R[J] ) THEN MAX_COST := COST.R1^.R[J];
    2 : IF( TEMP < COST.R2^.R[J] ) THEN MAX_COST := COST.R2^.R[J];
    3 : IF( TEMP < COST.R3^.R[J] ) THEN MAX_COST := COST.R3^.R[J];
    4 : IF( TEMP < COST.R4^.R[J] ) THEN MAX_COST := COST.R4^.R[J];
    5 : IF( TEMP < COST.R5^.R[J] ) THEN MAX_COST := COST.R5^.R[J];
    6 : IF( TEMP < COST.R6^.R[J] ) THEN MAX_COST := COST.R6^.R[J];
    7 : IF( TEMP < COST.R7^.R[J] ) THEN MAX_COST := COST.R7^.R[J];
    8 : IF( TEMP < COST.R8^.R[J] ) THEN MAX_COST := COST.R8^.R[J];
    9 : IF( TEMP < COST.R9^.R[J] ) THEN MAX_COST := COST.R9^.R[J];
    OTHERWISE
    BEGIN
      WRITELN(' Error - MAX_COST - DIM out of bounds');
      WRITELN(' CUT = ':7,CUT:3,',    COST.DIM = ':15,COST.DIM:3);
    END;
  END;
END;
{-------------------------------------------------------------------------}
PROCEDURE STORE_VAR( VAR LORD : PTR_TO_LINK_ORD;
          VAR DIM : INTEGER;
          SAVE : INTEGER;
          VAR J : INTEGER;
          VAR CUT : INTEGER);
BEGIN
```

AD-A117 948 SYSTEMS CONTROL TECHNOLOGY INC PALO ALTO CA F/G 9/2
ENHANCEMENTS AND ALGORITHMS FOR AVIONIC INFORMATION PROCESSING --ETC(U)
JUN 82 K DOTY, A LEMOINE, P MCENTIRE N62269-81-C-0477
UNCLASSIFIED NADC-81105-50 NL

2 OF 2
AD A
117948

END
DATE FILMED
08-82
DTIC

```
    IF( LORD <> NIL ) THEN
    BEGIN
      CASE DIM OF
        0 : LORD^.LINK^.I0^.I[J] := SAVE MOD 2;
        1 : LORD^.LINK^.I1^.I[J] := SAVE MOD 2;
        2 : LORD^.LINK^.I2^.I[J] := SAVE MOD 2;
        3 : LORD^.LINK^.I3^.I[J] := SAVE MOD 2;
        4 : LORD^.LINK^.I4^.I[J] := SAVE MOD 2;
        5 : LORD^.LINK^.I5^.I[J] := SAVE MOD 2;
        6 : LORD^.LINK^.I6^.I[J] := SAVE MOD 2;
        7 : LORD^.LINK^.I7^.I[J] := SAVE MOD 2;
        8 : LORD^.LINK^.I8^.I[J] := SAVE MOD 2;
        9 : LORD^.LINK^.I9^.I[J] := SAVE MOD 2;
        OTHERWISE
        BEGIN
          WRITELN(' Error - STORE_VAR - DIM out of bounds');
          WRITELN(' CUT = ':7,CUT:3,',    DIM = ':10,DIM:3);
        END;
      END;
      SAVE := SAVE DIV 2;
      STORE_VAR( LORD^.NEXT, DIM, SAVE, J, CUT);
    END;
  END;
  {-------------------------------------------------------------------------}
  PROCEDURE COMPUTE_J_COST( VAR LROOT : PTR_TO_LINK_REC;
            VAR J_COST : COST_REC;
            VAR J_PREV_COST : COST_REC;
            VAR K_COST : COST_REC;
            VAR INTERNALS : VAR_ORD;
            VAR CUR_NODE : PTR_TO_NODE_REC);
  VAR
    J_IN,SAVE_IN,J_EX,J_CUT,J_PREV,LIMIT_IN,LIMIT_EX,BIT_SUM,K : INTEGER;
    TEMP_MAX,TEMP_MIN : REAL;
  BEGIN
    IF((INTERNALS.DIM < 0) OR (INTERNALS.DIM > MAX_DIM)) THEN
    BEGIN
      WRITELN(' Error - COMPUTE_J_COST - INTERNALS.DIM out of bounds');
      WRITELN(' CUT = ':7,CUR_NODE^.NAME:3,
        ',    INTERNALS.DIM = ':20, INTERNALS.DIM:3);
    END;
    IF((J_COST.DIM < 0) OR (J_COST.DIM > MAX_DIM)) THEN
    BEGIN
      WRITELN(' Error - COMPUTE_J_COST - J_COST.DIM out of bounds');
      WRITELN(' CUT = ':7,CUR_NODE^.NAME:3,
        ',    J_COST.DIM = ':17, J_COST.DIM:3);
    END;
    CASE J_COST.DIM OF
      0 : NEW(J_COST.R0);
      1 : NEW(J_COST.R1);
      2 : NEW(J_COST.R2);
      3 : NEW(J_COST.R3);
      4 : NEW(J_COST.R4);
      5 : NEW(J_COST.R5);
      6 : NEW(J_COST.R6);
      7 : NEW(J_COST.R7);
      8 : NEW(J_COST.R8);
      9 : NEW(J_COST.R9);
      OTHERWISE
      BEGIN
        WRITELN(' Error - COMPUTE_J_COST - J_COST.DIM out of range');
        WRITELN(' CUT = ':7,CUR_NODE^.NAME:3,
```

```
        ',   J_COST.DIM = ':17, J_COST.DIM:3);
    END;
  END;
  LIMIT_IN := (2**INTERNALS.DIM) - 1;
  LIMIT_EX := (2**J_COST.DIM) - 1;
  FOR J_EX := 0 TO LIMIT_EX DO
  BEGIN
    TEMP_MIN := INFINITY;
    SAVE_IN := 0;
    FOR J_IN := 0 TO LIMIT_IN DO
    BEGIN
      RECORD_VAR( INTERNALS.LORD, J_IN);
      RECORD_VAR( J_COST.LORD, J_EX);
      RETRIEVE_VAR( K_COST.LORD, J_CUT);
      RETRIEVE_VAR( J_PREV_COST.LORD, J_PREV);
      CASE CUR_NODE^.KIND OF
        PRO : BEGIN
                TEMP_MAX := MAX_COST( 0.0, J_PREV_COST, J_PREV, CUR_NODE^.NAME);
                TEMP_MAX := MAX_COST( TEMP_MAX, K_COST, J_CUT, CUR_NODE^.NAME);
              END;
        MODUL : BEGIN
                  BIT_SUM := 0;
                  FOR K := 1 TO K_COST.DIM DO
                  BEGIN
                    BIT_SUM := BIT_SUM + (J_CUT MOD 2);
                    J_CUT := J_CUT DIV 2;
                  END;
                  IF( BIT_SUM = 1 ) THEN
                    TEMP_MAX := MAX_COST(0.0, J_PREV_COST, J_PREV, CUR_NODE^.NAME)
                  ELSE
                    TEMP_MAX := INFINITY;
                END;
        OTHERWISE
        BEGIN
          WRITELN(' Error - COMPUTE_J_COST - KIND <> PRO or MODUL');
          WRITELN(' CUT = ':7, CUR_NODE^.NAME:3,
            ',   CUR_NODE^.KIND = ':20, CUR_NODE^.KIND :5);
        END;
      END;
      IF( TEMP_MIN > TEMP_MAX ) THEN
      BEGIN
        TEMP_MIN := TEMP_MAX;
        SAVE_IN := J_IN;
      END;
      STORE_VAR( INTERNALS.LORD, INTERNALS.DIM, SAVE_IN, J_EX, CUR_NODE^.NAME);
      CASE J_COST.DIM OF
        0 : J_COST.R0^.R[J_EX] := TEMP_MIN;
        1 : J_COST.R1^.R[J_EX] := TEMP_MIN;
        2 : J_COST.R2^.R[J_EX] := TEMP_MIN;
        3 : J_COST.R3^.R[J_EX] := TEMP_MIN;
        4 : J_COST.R4^.R[J_EX] := TEMP_MIN;
        5 : J_COST.R5^.R[J_EX] := TEMP_MIN;
        6 : J_COST.R6^.R[J_EX] := TEMP_MIN;
        7 : J_COST.R7^.R[J_EX] := TEMP_MIN;
        8 : J_COST.R8^.R[J_EX] := TEMP_MIN;
        9 : J_COST.R9^.R[J_EX] := TEMP_MIN;
        OTHERWISE
        BEGIN
          WRITELN(' Error - COMPUTE_J_COST - J_COST.DIM out of range');
          WRITELN(' CUT = ':7, CUR_NODE^.NAME:3,
            ',   J_COST.DIM = ':16, J_COST.DIM:3);
```

```
          END;
        END;
      END;
    END;
  END;
{------------------------------------------------------------------------}
END.
```

```
MODULE WRITEDATA(INPUT,OUTPUT,INFILE,OUTFILE);

{------ Declarations ------}

%INCLUDE 'SDP2.DEC'

{-------------------------------------------------------------}
PROCEDURE WRITE_NODES( VAR NROOT : PTR_TO_NODE_REC);
BEGIN
  IF( NROOT <> NIL ) THEN
  BEGIN
    WITH NROOT^ DO
    BEGIN
      CASE KIND OF
        PRO :   WRITELN(OUTFILE,' Processor :  ':13,NAME:2,
                  ',      Mem capacity :  ':21,MEM_CAP:8,
                  ',           Speed :  ':18,SPEED:8:2);
        MODUL : WRITELN(OUTFILE,'    Module :  ':13,NAME:2,
                  ',  Mem requirement :  ':21,MEM_REQ:8,
                  ',  Instructions :  ':18,INSTR:8,
                  ',  Time Constraint :  ':21,TIME_CONSTRAINT:8:2);
        OTHERWISE
        BEGIN
          WRITELN(' Error - WRITE_NODES - KIND <> PRO or MODUL');
          WRITELN(' node # = ':10,NAME:3,',  KIND = ':10,KIND:5);
        END;
      END;
    END;
    WRITE_NODES( NROOT^.NEXT );
  END;
END;
{-------------------------------------------------------------}
PROCEDURE WRITE_LINKS( VAR LROOT : PTR_TO_LINK_REC);
BEGIN
  IF( LROOT <> NIL ) THEN
  BEGIN
    WITH LROOT^ DO
      WRITELN(OUTFILE,' Initially permissible link from ':33,
        NODE1:3,' to ':4,NODE2:3);
    WRITE_LINKS( LROOT^.NEXT );
  END;
END;
{-------------------------------------------------------------}
PROCEDURE WRITE_FIRMLINKS( VAR LROOT : PTR_TO_LINK_REC;
          VAR NAME : INTEGER);
BEGIN
  IF( LROOT <> NIL ) THEN
  BEGIN
    WITH LROOT^ DO
    BEGIN
      IF(( NODE1 = NAME ) AND ( CONNECT = 1 )) THEN
        WRITE(OUTFILE,',':1,NODE2:3);
      IF(( NODE2 = NAME ) AND ( CONNECT = 1 )) THEN
        WRITE(OUTFILE,',':1,NODE1:3);
      WRITE_FIRMLINKS( NEXT, NAME );
    END;
  END;
END;
{-------------------------------------------------------------}
PROCEDURE WRITE_PRO( VAR NROOT : PTR_TO_NODE_REC;
          VAR LROOT : PTR_TO_LINK_REC);
```

```
BEGIN
  IF( NROOT <> NIL ) THEN
  BEGIN
    WITH NROOT^ DO
    BEGIN
      CASE KIND OF
        PRO   : BEGIN
                  IF( TIME_USE > 0.99*INFINITY ) THEN
                    WRITELN(OUTFILE,' Processor ':11,NAME:3,
                    ', amount of time is  infinite ':30,
                    ', amount of memory is ':22,MEM_USE:8)
                  ELSE
                    WRITELN(OUTFILE,' Processor ':11,NAME:3,
                    ', amount of time is ':20,TIME_USE:10:2,
                    ', amount of memory is ':22,MEM_USE:8);
                  WRITE(OUTFILE,' Firm links are ... ':19);
                  WRITE_FIRMLINKS(LROOT,NAME);
                  WRITELN(OUTFILE); WRITELN(OUTFILE);
                END;
        MODUL : ;
        OTHERWISE
          WRITELN(' Error - WRITE_PRO - invalid NROOT^.KIND')
      END;
      WRITE_PRO(NEXT,LROOT);
    END;
  END;
END;
{-----------------------------------------------------------------}
PROCEDURE WRITE_COM_LINKS( VAR CROOT : PTR_TO_COM_REC );
BEGIN
  IF( CROOT <> NIL ) THEN
  BEGIN
    WITH CROOT^ DO
    BEGIN
      WRITELN(OUTFILE,' module ':8,MOD1:3,' must communicate with':22,
        ' module ':8,MOD2:3,' with delay ':12,DELAY:8.2);
    END;
    WRITE_COM_LINKS( CROOT^.NEXT );
  END;
END;
{-----------------------------------------------------------------}
PROCEDURE WRITEDATA( VAR NROOT : PTR_TO_NODE_REC;
          VAR LROOT : PTR_TO_LINK_REC;
          VAR CROOT : PTR_TO_COM_REC);
BEGIN
  WRITELN(OUTFILE);
  WRITE_NODES( NROOT );
  WRITELN(OUTFILE);
  WRITE_LINKS( LROOT );
  WRITELN(OUTFILE);
  WRITE_COM_LINKS( CROOT );
  IF( CROOT = NIL ) THEN
    WRITELN(OUTFILE,' There are no module communication constraints.');
  WRITELN(OUTFILE);
  WRITE_PRO( NROOT, LROOT );
END;
{-----------------------------------------------------------------}
END.
```

```
(------ Declarations ------)
CONST
  MAX_DIM = 9;
  INFINITY = 1.7E38;
TYPE
  SUB1 = 1..9;
  SUB2 = 1..41;
  NODE_KIND = (PRO,MODUL);
  PTR_TO_NODE_REC = ^NODE_REC;
  NODE_REC = RECORD
               NAME : INTEGER;
               NEXT  : PTR_TO_NODE_REC;
               CASE KIND : NODE_KIND OF
                 PRO   : (MEM_CAP : INTEGER;
                          SPEED   : REAL;
                          TIME_USE: REAL;
                          MEM_USE : INTEGER);
                 MODUL : (MEM_REQ : INTEGER;
                          INSTR: INTEGER;
                          TIME_CONSTRAINT : REAL);
             END; { NODE_REC }
  B = 0..1;
  DIM_RANGE = -1..MAX_DIM;

  PTR_BINR0_REC = ^BINR0_REC;
  BINR0_REC = RECORD R : ARRAY [0..0] OF REAL; END;
  PTR_BINR1_REC = ^BINR1_REC;
  BINR1_REC = RECORD R : ARRAY [0..1] OF REAL; END;
  PTR_BINR2_REC = ^BINR2_REC;
  BINR2_REC = RECORD R : ARRAY [0..3] OF REAL; END;
  PTR_BINR3_REC = ^BINR3_REC;
  BINR3_REC = RECORD R : ARRAY [0..7] OF REAL; END;
  PTR_BINR4_REC = ^BINR4_REC;
  BINR4_REC = RECORD R : ARRAY [0..15] OF REAL; END;
  PTR_BINR5_REC = ^BINR5_REC;
  BINR5_REC = RECORD R : ARRAY [0..31] OF REAL; END;
  PTR_BINR6_REC = ^BINR6_REC;
  BINR6_REC = RECORD R : ARRAY [0..63] OF REAL; END;
  PTR_BINR7_REC = ^BINR7_REC;
  BINR7_REC = RECORD R : ARRAY [0..127] OF REAL; END;
  PTR_BINR8_REC = ^BINR8_REC;
  BINR8_REC = RECORD R : ARRAY [0..255] OF REAL; END;
  PTR_BINR9_REC = ^BINR9_REC;
  BINR9_REC = RECORD R : ARRAY [0..511] OF REAL; END;

  PTR_BINI0_REC = ^BINI0_REC;
  BINI0_REC = RECORD I : ARRAY [0..0] OF B; END;
  PTR_BINI1_REC = ^BINI1_REC;
  BINI1_REC = RECORD I : ARRAY [0..1] OF B; END;
  PTR_BINI2_REC = ^BINI2_REC;
  BINI2_REC = RECORD I : ARRAY [0..3] OF B; END;
  PTR_BINI3_REC = ^BINI3_REC;
  BINI3_REC = RECORD I : ARRAY [0..7] OF B; END;
  PTR_BINI4_REC = ^BINI4_REC;
  BINI4_REC = RECORD I : ARRAY [0..15] OF B; END;
  PTR_BINI5_REC = ^BINI5_REC;
  BINI5_REC = RECORD I : ARRAY [0..31] OF B; END;
  PTR_BINI6_REC = ^BINI6_REC;
  BINI6_REC = RECORD I : ARRAY [0..63] OF B; END;
  PTR_BINI7_REC = ^BINI7_REC;
  BINI7_REC = RECORD I : ARRAY [0..127] OF B; END;
```

```
    PTR_BINI8_REC = ^BINI8_REC;
    BINI8_REC = RECORD I : ARRAY [0..255] OF B; END;
    PTR_BINI9_REC = ^BINI9_REC;
    BINI9_REC = RECORD I : ARRAY [0..511] OF B; END;

    PTR_TO_LINK_REC = ^LINK_REC;
    LINK_REC = RECORD
                 NODE1 : INTEGER;
                 NODE2 : INTEGER;
                 CONNECT : INTEGER;
                 NEXT : PTR_TO_LINK_REC;
                 CASE DIM : DIM_RANGE OF
                   -1      : ( );
                    0      : (I0 : PTR_BINI0_REC);
                    1      : (I1 : PTR_BINI1_REC);
                    2      : (I2 : PTR_BINI2_REC);
                    3      : (I3 : PTR_BINI3_REC);
                    4      : (I4 : PTR_BINI4_REC);
                    5      : (I5 : PTR_BINI5_REC);
                    6      : (I6 : PTR_BINI6_REC);
                    7      : (I7 : PTR_BINI7_REC);
                    8      : (I8 : PTR_BINI8_REC);
                    9      : (I9 : PTR_BINI9_REC);
               END; { LINK_REC }
    PTR_TO_LINK_ORD = ^LINK_ORD;
    LINK_ORD = RECORD
                 LINK : PTR_TO_LINK_REC;
                 NODE : PTR_TO_NODE_REC;
                 NEXT : PTR_TO_LINK_ORD;
               END;
    VAR_ORD = RECORD
                DIM: INTEGER;
                LORD : PTR_TO_LINK_ORD;
              END;
    COST_REC = RECORD
                 LORD : PTR_TO_LINK_ORD;
                 CASE DIM : DIM_RANGE OF
                   -1      : ( );
                    0      : (R0 : PTR_BINR0_REC);
                    1      : (R1 : PTR_BINR1_REC);
                    2      : (R2 : PTR_BINR2_REC);
                    3      : (R3 : PTR_BINR3_REC);
                    4      : (R4 : PTR_BINR4_REC);
                    5      : (R5 : PTR_BINR5_REC);
                    6      : (R6 : PTR_BINR6_REC);
                    7      : (R7 : PTR_BINR7_REC);
                    8      : (R8 : PTR_BINR8_REC);
                    9      : (R9 : PTR_BINR9_REC);
               END; { COST_REC }
  PTR_TO_COM_REC = ^COM_REC;
  COM_REC = RECORD
              MOD1 : INTEGER;
              PMOD1 : PTR_TO_LINK_REC;
              MOD2 : INTEGER;
              PMOD2 : PTR_TO_LINK_REC;
              DELAY : REAL;
              CONSIDER : BOOLEAN;
              NEXT : PTR_TO_COM_REC;
            END;
  PTR_TO_COM_ORD = ^COM_ORD;
  COM_ORD = RECORD
```

```
    COM : PTR_TO_COM_REC;
    NEXT : PTR_TO_COM_ORD;
  END;

VAR
  LROOT : PTR_TO_LINK_REC;
  NROOT : PTR_TO_NODE_REC;
  CROOT : PTR_TO_COM_REC;
  J_COST : COST_REC;
  CUT : INTEGER;
  PROMPT : PACKED ARRAY [SUB2] OF CHAR;
  INFILE : TEXT;
  OUTFILE : TEXT;
```

SAMPLE INPUT AND OUTPUT
FOR
SDP PROGRAM

```
28
1  MODULE      45   79     60.0
2  MODULE      24   99     60.0
3  PROCESSOR  150  4.0
4  MODULE      62   45     70.0
5  MODULE      51   32     80.0
6  PROCESSOR   64  3.0
7  MODULE      53   66     40.0
8  MODULE      37   44     60.0
9  PROCESSOR  150  4.0
10 MODULE      59   47     70.0
11 MODULE      44   63     90.0
12 MODULE      32   89     40.0
13 PROCESSOR  150  2.0
14 MODULE      57   92     50.0
15 MODULE      21   20     70.0
16 MODULE      73   81     80.0
17 PROCESSOR  192  4.0
18 MODULE      67   29     50.0
19 MODULE      71   84     60.0
20 PROCESSOR  150  3.0
21 MODULE      30   37     40.0
22 MODULE      72   84     90.0
23 MODULE      53   72     70.0
24 PROCESSOR  256  2.0
25 MODULE      55   81     60.0
26 PROCESSOR  200  4.0
27 MODULE      53   57     60.0
28 MODULE      18   21     80.0
42
1 3
1 6
2 3
2 6
4 3
4 6
5 3
5 9
7 3
7 9
8 6
8 9
8 13
10 6
10 13
11 9
11 13
12 9
12 13
14 13
14 17
15 13
15 20
16 13
16 17
16 20
18 17
18 20
19 17
19 20
21 17
```

21	24	
22	20	
22	24	
23	20	
23	24	
25	24	
25	26	
27	24	
27	26	
28	24	
28	26	
10		
1	5	0
2	7	0
5	10	0
7	11	0
8	12	0
12	16	0
14	18	0
16	21	0
19	25	0
22	28	0

Maximum cost is 66.00

Module :	1.	Mem requirement :	45.	Instructions :	79.	Time Constraint :	60.00	
Module :	2.	Mem requirement :	24.	Instructions :	99.	Time Constraint :	60.00	
Processor :	3.	Mem capacity :	150.	Speed :	4.00			
Module :	4.	Mem requirement :	62.	Instructions :	45.	Time Constraint :	70.00	
Module :	5.	Mem requirement :	51.	Instructions :	32.	Time Constraint :	80.00	
Processor :	6.	Mem capacity :	64.	Speed :	3.00			
Module :	7.	Mem requirement :	53.	Instructions :	66.	Time Constraint :	40.00	
Module :	8.	Mem requirement :	37.	Instructions :	44.	Time Constraint :	60.00	
Processor :	9.	Mem capacity :	150.	Speed :	4.00			
Module :	10.	Mem requirement :	59.	Instructions :	47.	Time Constraint :	70.00	
Module :	11.	Mem requirement :	44.	Instructions :	63.	Time Constraint :	90.00	
Module :	12.	Mem requirement :	32.	Instructions :	89.	Time Constraint :	40.00	
Processor :	13.	Mem capacity :	150.	Speed :	2.00			
Module :	14.	Mem requirement :	57.	Instructions :	92.	Time Constraint :	50.00	
Module :	15.	Mem requirement :	21.	Instructions :	20.	Time Constraint :	70.00	
Module :	16.	Mem requirement :	73.	Instructions :	81.	Time Constraint :	80.00	
Processor :	17.	Mem capacity :	192.	Speed :	4.00			
Module :	18.	Mem requirement :	67.	Instructions :	29.	Time Constraint :	50.00	
Module :	19.	Mem requirement :	71.	Instructions :	84.	Time Constraint :	60.00	
Processor :	20.	Mem capacity :	150.	Speed :	3.00			
Module :	21.	Mem requirement :	30.	Instructions :	37.	Time Constraint :	40.00	
Module :	22.	Mem requirement :	72.	Instructions :	84.	Time Constraint :	90.00	
Module :	23.	Mem requirement :	53.	Instructions :	72.	Time Constraint :	70.00	
Processor :	24.	Mem capacity :	256.	Speed :	2.00			
Module :	25.	Mem requirement :	55.	Instructions :	81.	Time Constraint :	60.00	
Processor :	26.	Mem capacity :	200.	Speed :	4.00			
Module :	27.	Mem requirement :	53.	Instructions :	57.	Time Constraint :	60.00	
Module :	28.	Mem requirement :	18.	Instructions :	21.	Time Constraint :	80.00	

Initially permissible link from 1 to 3
Initially permissible link from 1 to 6
Initially permissible link from 2 to 3
Initially permissible link from 2 to 6
Initially permissible link from 3 to 4
Initially permissible link from 4 to 6
Initially permissible link from 3 to 5
Initially permissible link from 5 to 9
Initially permissible link from 3 to 7
Initially permissible link from 7 to 9
Initially permissible link from 6 to 8
Initially permissible link from 8 to 9
Initially permissible link from 8 to 13
Initially permissible link from 6 to 10
Initially permissible link from 10 to 13
Initially permissible link from 9 to 11
Initially permissible link from 11 to 13
Initially permissible link from 9 to 12
Initially permissible link from 12 to 13
Initially permissible link from 13 to 14
Initially permissible link from 14 to 17
Initially permissible link from 13 to 15
Initially permissible link from 15 to 20
Initially permissible link from 13 to 16
Initially permissible link from 16 to 17
Initially permissible link from 16 to 20
Initially permissible link from 17 to 18
Initially permissible link from 18 to 20
Initially permissible link from 17 to 19
Initially permissible link from 19 to 20

```
Initially permissible link from  17 to  21
Initially permissible link from  21 to  24
Initially permissible link from  20 to  22
Initially permissible link from  22 to  24
Initially permissible link from  20 to  23
Initially permissible link from  23 to  26
Initially permissible link from  24 to  25
Initially permissible link from  25 to  26
Initially permissible link from  24 to  27
Initially permissible link from  26 to  27
Initially permissible link from  24 to  28
Initially permissible link from  26 to  28

module   1 must communicate with module   5 with delay      5.00
module   2 must communicate with module   7 with delay      4.00
module   5 must communicate with module  10 with delay      3.00
module   7 must communicate with module  11 with delay      6.00
module   8 must communicate with module  12 with delay      2.00
module  12 must communicate with module  16 with delay      4.00
module  14 must communicate with module  18 with delay      5.00
module  16 must communicate with module  21 with delay      6.00
module  19 must communicate with module  25 with delay      2.00
module  22 must communicate with module  28 with delay      4.00

Processor   3, amount of time is      58.50, amount of memory is      139
Firm links are ...,  2,  4,  7

Processor   6, amount of time is      31.33, amount of memory is       45
Firm links are ...,  1

Processor   9, amount of time is      66.00, amount of memory is      127
Firm links are ...,  5, 11, 12

Processor  13, amount of time is      60.50, amount of memory is      117
Firm links are ...,  8, 10, 15

Processor  17, amount of time is      61.50, amount of memory is      160
Firm links are ..., 14, 16, 21

Processor  20, amount of time is      44.67, amount of memory is      138
Firm links are ..., 18, 19

Processor  24, amount of time is      52.50, amount of memory is       90
Firm links are ..., 22, 28

Processor  26, amount of time is      54.50, amount of memory is      161
Firm links are ..., 23, 25, 27
```

APPENDIX B

NETEV PROGRAM
TO
EVALUATE COMPUTER NETWORKS

```
      PROGRAM ARCH
C
C EVALUATES DIFFERENT ARCHITECTURES
C
      INCLUDE 'DIMEN.FOR'
      COMMON/DISTAN/NNODE,DIST(MNOD,MNOD)
      COMMON/FLAGS/IPRFG
      REAL SHORT(MNOD,MNOD)
      REAL WK1(MNOD,MNOD),WK2(MNOD,MNOD),WK3(MNOD,MNOD)
C
C READ INPUT
      CALL RDINPUT
C
C FIND SHORTEST PATH BETWEEN ALL PAIRS OF NODES & PRINT THE PATH MATRIX
      CALL PATH(MNOD,NNODE,DIST,SHORT)
      IF(IPRFG .EQ. 1)  CALL PRNTP(MNOD,NNODE,SHORT,WK3)
C
C FIND MAXIMUM DISTANCE BETWEEN TWO NODES
      CALL MAXDIS(MNOD,NNODE,SHORT,PMAX)
      WRITE(LW,160) PMAX
160   FORMAT('0','MAX DISTANCE= ',F8.4)
C
C COMPUTE AVERAGE DISTANCE BETWEEN TWO NODES
      CALL AVGDIS(MNOD,NNODE,SHORT,AVG)
      WRITE(LW,170) AVG
170   FORMAT('0','AVG DISTANCE= ',F8.4)
C
C ANALYZE LINK FAILURES
      CALL LKFAIL(MNOD,NNODE,DIST,WK1,WK2,WK3)
C
C ANALYZE NODE FAILURES
      CALL NDFAIL(MNOD,NNODE,DIST,WK1,WK2,WK3)
C
C
      STOP
      END
```

```
      SUBROUTINE RDINPUT
C
C READS INPUT TO ARCH
C
      INCLUDE 'DIMEN.FOR'
      COMMON/DISTAN/NNODE,DIST(MNOD,MNOD)
      COMMON/FLAGS/IPRFG
C
      READ(LR,10) NNODE
10    FORMAT(I3)
      WRITE(LW,20) NNODE
20    FORMAT(' ','NUMBER OF NODES IN NETWORK= ',I3)
C
C INITIALIZE DISTANCE ARRAY TO INFINITY.  DIAGONAL ELEMENTS ARE ZERO.
      DO 40 I=1,NNODE
      DO 40 J=1,NNODE
        DIST(I,J) = 1.E+30
40    CONTINUE
      DO 50 I=1,NNODE
        DIST(I,I) = 0.0
50    CONTINUE
C
C READ # DISTANCES TO BE READ.  THEN READ THE DISTANCES.
      READ(LR,60) NDIST
60    FORMAT(I4)
      WRITE(LW,70) NDIST
70    FORMAT('0','NUMBER DISTANCES TO BE READ= ',I4)
      WRITE(LW,75)
75    FORMAT('0',5X,'FROM  TO    DISTANCES')
C IF FLAG SET, READ DISTANCES.  ELSE ASSUME DISTANCE OF 1
      READ(LR,76) IDISFG
76    FORMAT(I1)
      IF (IDISFG .EQ. 1) THEN
        DO 100 I=1,NDIST
         READ(LR,80) JF,JT,DIST(JF,JT)
80         FORMAT(2(I3,1X),F8.4)
           DIST(JT,JF) = DIST(JF,JT)
           WRITE(LW,90) JF,JT,DIST(JF,JT)
90         FORMAT(5X,I3,2X,I3,4X,F8.4)
100     CONTINUE
      ELSE
        DO 200 I=1,NDIST
          READ(LR,120) JF,JT
120       FORMAT(2(I3,1X))
          DIST(JF,JT) = 1.0
          DIST(JT,JF) = 1.0
          WRITE(LW,90) JF,JT,DIST(JF,JT)
200     CONTINUE
      END IF
C
C FLAG FOR PRINTING
      READ(LR,300) IPRFG
300   FORMAT(I1)
C
C
      RETURN
      END
```

```
      SUBROUTINE PATH(NDIM,N,A,B)
C
C FINDS THE SHORTEST PATH BETWEEN ALL PAIRS OF NODES USING THE
C FLOYD-WARSHALL ALGORITM
C
      INCLUDE 'DIMEN.FOR'
      REAL A(NDIM,NDIM),B(NDIM,NDIM)
C
C COPY DISTANCE ARRAY INTO FIRST U
      DO 10 I=1,N
      DO 10 J=1,N
        B(I,J) = A(I,J)
10    CONTINUE
C
C COMPUTE NEW U'S. DETERMINE SHORTEST PATH BETWEEN NODES, GIVEN
C ABILITY TO TRAVERSE ONE MORE NODE EACH LOOP.
      DO 100 M=1,N
        DO 50 JF=1,N-1
        DO 40 JT=JF+1,N
          IF (M .EQ. JF) GO TO 50
          IF (M .EQ. JT) GO TO 40
          B(JF,JT) = MIN(B(JF,JT),B(JF,M)+B(M,JT))
          B(JT,JF) = B(JF,JT)
40        CONTINUE
50        CONTINUE
100   CONTINUE
C
C
      RETURN
      END
```

```
      SUBROUTINE MAXDIS(NDIM,N,B,PMAX)
C
C FINDS MAXIMUM DISTANCE BETWEEN NODES
C
      INCLUDE 'DIMEN.FOR'
      REAL B(NDIM,NDIM)
C
      PMAX = -1.0
      DO 10 I1=1,N-1
      DO 10 I2=I1+1,N
        PMAX = MAX(PMAX,B(I1,I2))
10    CONTINUE
C
C
      RETURN
      END
```

```
      SUBROUTINE AVGDIS(NDIM,N,B,AVG)
C
C COMPUTES AVERAGE DISTANCE BETWEEN NODES
C
      INCLUDE 'DIMEN.FOR'
      REAL B(NDIM,NDIM)
C
      AVG = 0.0
      DO 10 I1=1,N-1
      DO 10 I2=I1+1,N
        AVG = AVG + B(I1,I2)
10    CONTINUE
C
      AVG = AVG/((N*(N-1))/2)
C
C
      RETURN
      END
```

```
      SUBROUTINE LKFAIL(NDIM,N,A,TEST,OUT,WK)
C
C ANALYZES LINK FAILURES.  COMPUTES MAX DISTANCE AND AVERAGE DISTANCE
C BETWEEN TWO NODES FOR EACH LINK.  THEN COMPUTES THE AVERAGE
C OF THESE TWO STATISTICS.
C
      INCLUDE 'DIMEN.FOR'
      REAL A(NDIM,NDIM),OUT(NDIM,NDIM),TEST(NDIM,NDIM),WK(NDIM,NDIM)
      REAL FIN(MLIN)
C
C INITIALIZE
      NL = 0
      AVGMAX = 0.0
      AVGAVG = 0.0
      NUM1 = 0
      AVGFIN = 0.0
      VARFIN = 0.0
      DO 10 I1=1,N
      DO 10 I2=1,N
        TEST(I1,I2) = A(I1,I2)
10    CONTINUE
C
C LOOP THROUGH TEST ARRAY.  DELETE EACH LINK, ANALYZE STATS, THEN
C RESTORE LINK AND REPEAT FOR NEXT LINK.
      DO 100 I1=1,N-1
      DO 100 I2=I1+1,N
C
C FIND LINK
        IF (TEST(I1,I2) .GE. 1.E+30) GO TO 100
C
C DELETE LINK, INCREMENT # LINKS.
        TEST(I1,I2) = 1.E+30
        TEST(I2,I1) = 1.E+30
        NL = NL + 1
C
C SHORTEST PATH WITH THIS LINK DELETED.
        CALL PATH(NDIM,N,TEST,OUT)
C
C MAXIMUM DISTANCE BETWEEN TWO NODES.  KEEP RUNNING SUM.
        CALL MAXDIS(NDIM,N,OUT,TMAX)
        AVGMAX = AVGMAX + TMAX
C
C AVERAGE DISTANCE BETWEEN TWO NODES.  KEEP RUNNING SUM.
        CALL AVGDIS(NDIM,N,OUT,TAVG)
        AVGAVG = AVGAVG + TAVG
C
C HOW MANY ELEMENTS OF SHORTEST PATH ARE LESS THAN INFINITY
        CALL FINITE(NDIM,N,OUT,FIN(NL))
C
C RESTORE LINK
        TEST(I1,I2) = A(I1,I2)
        TEST(I2,I1) = A(I2,I1)
C
C
100   CONTINUE
C
C COMPUTE AVERAGE OF MAX DISTANCES AND AVERAGE OF AVG DISTANCES.
      AVGMAX = AVGMAX/FLOAT(NL)
      AVGAVG = AVGAVG/FLOAT(NL)
C
C PRINT AVERAGES
```

```
      WRITE(LW,200) AVGMAX
200   FORMAT('0','AVERAGE MAX DISTANCE IF A LINK FAILS= ',E11.4)
      WRITE(LW,300) AVGAVG
300   FORMAT('0','AVERAGE AVG DISTANCE IF A LINK FAILS= ',E11.4)
C
C STATISTICS ON NUMBER FINITE DISTANCES - # LESS THAN 1;
C AVERAGE, VARIANCE
      DO 400 I=1,NL
        IF (ABS(1.-FIN(I)) .LT. 1.E-04) NUM1 = NUM1 + 1
        AVGFIN = AVGFIN + FIN(I)
400   CONTINUE
      FRAC1 = FLOAT(NUM1)/FLOAT(NL)
      AVGFIN = AVGFIN/FLOAT(NL)
      DO 500 I=1,NL
        VARFIN = VARFIN + (FIN(I) - AVGFIN)**2
500   CONTINUE
      VARFIN = VARFIN/FLOAT(NL)
C
C PRINT THESE STATS
      WRITE(LW,600) FRAC1
600   FORMAT('0','FRACTION OF LINK FAILURES WHERE DISTANCES ',
     *       'ALL FINITE= ',F10.4)
      WRITE(LW,610) AVGFIN,VARFIN
610   FORMAT('0','AVERAGE FRACTION CONNECTED NODES, GIVEN ',
     *       'LINK FAILURE= ',F10.4,5X,'VARIANCE= ',F10.4)
C
C
      RETURN
      END
```

```
      SUBROUTINE NDFAIL(NDIM,N,A,TEST,OUT,WK)
C
C ANALYZES NODE FAILURES
C
      INCLUDE 'DIMEN.FOR'
      REAL A(NDIM,NDIM),TEST(NDIM,NDIM),OUT(NDIM,NDIM),WK(NDIM,NDIM)
C
C INITIALIZE
      AVGMAX = 0.0
      AVGAVG = 0.0
      NM1 = N - 1
      DO 10 I=1,NM1
      DO 10 J=1,NM1
        TEST(I,J) = 0.0
10    CONTINUE
C
C LOOP THROUGH ALL NODES.  DELETE EACH, ANALYZE STATS, RESTORE & REPEAT.
      DO 1000 IN=1,N
        M1 = 1
        M2 = 1
        DO 50 I1=1,NM1
          IF (IN .EQ. I1) M1 = M1+1
          DO 40 I2=I1,NM1
            IF (IN .EQ. I2) M2 = M2 + 1
            TEST(I1,I2) = A(M1,M2)
            TEST(I2,I1) = A(M1,M2)
            M2 = M2 + 1
40        CONTINUE
          M1 = M1 + 1
          M2 = M1
50      CONTINUE
C
C SHORTEST PATH FOR THIS NEW MATRIX WITH NODE DELETED.
        CALL PATH(NDIM,NM1,TEST,OUT)
C
C MAXIMUM DISTANCE.  KEEP RUNNING SUM.
        CALL MAXDIS(NDIM,NM1,OUT,TMAX)
        AVGMAX = AVGMAX + TMAX
C
C AVERAGE DISTANCE.  KEEP RUNNING SUM.
        CALL AVGDIS(NDIM,NM1,OUT,TAVG)
        AVGAVG = AVGAVG + TAVG
C
C
1000  CONTINUE
C
C COMPUTE AVERAGE OF MAX DISTANCES AND AVERAGE OF AVG DISTANCES.
      AVGMAX = AVGMAX/FLOAT(N)
      AVGAVG = AVGAVG/FLOAT(N)
C
C PRINT AVERAGES
      WRITE(LW,2000) AVGMAX
2000  FORMAT('0','AVERAGE MAX DISTANCE IF A NODE FAILS= ',E11.4)
      WRITE(LW,2100) AVGAVG
2100  FORMAT('0','AVERAGE AVG DISTANCE IF A NODE FAILS= ',E11.4)
C
C
      RETURN
      END
```

```
      SUBROUTINE PRNTP(NDIM,N,P,PR)
C
C PRINTS THE PATH MATRIX PASSED TO IT.
C
      INCLUDE 'DIMEN.FOR'
      REAL P(NDIM,NDIM),PR(NDIM,NDIM)
C
C COPY P TO PR FOR PRINTING.  DON'T WANT TO DESTROY P.
      DO 10 I=1,N
      DO 10 J=1,N
        PR(I,J) = P(I,J)
10    CONTINUE
C
C PRINT -1 IF DISTANCE IS INFINITY.
      DO 20 I=1,N
      DO 20 J=1,N
        IF (PR(I,J) .GE. 1.E+30) PR(I,J) = -1.0
20    CONTINUE
C
C PRINT
      WRITE(LW,110)
110   FORMAT('0',17X,'SHORTEST PATH')
      WRITE(LW,120)
120   FORMAT('0',22X,'TO NODE')
      WRITE(LW,125)
125   FORMAT(5X,'FROM NODE')
      DO 150 JF=1,N
        WRITE(LW,140) (PR(JF,JT),JT=1,N)
140     FORMAT(17X,20(F3.0,2X),/)
150   CONTINUE
C
C
      RETURN
      END
```

```
      SUBROUTINE FINITE(NDIM,N,B,FRAC)
C
C COMPUTES FRACTION OF ELEMENTS LESS THAN INFINITY
C
      INCLUDE 'DIMEN.FOR'
      REAL B(NDIM,NDIM)
C
      IFR = 0
      DO 10 I1=1,N-1
      DO 10 I2=I1+1,N
        IF (B(I1,I2) .LT. 1.E+30) IFR = IFR + 1
10    CONTINUE
      NUM = (N*(N-1))/2
      FN = FLOAT(NUM)
      FIFR = FLOAT(IFR)
      FRAC = FIFR/FN
C
C
      RETURN
      END
```

APPENDIX C

ROUTING ALGORITHM FOR DE BRUIJN GRAPHS

To describe the routing algorithm, the representation in [37] will be used. Each node is represented as a vector of k symbols, each taking values from 1 to d/2 (assuming d is even). Node

$$(v_1,v_2,\ldots,v_k)$$

is connected to all nodes of the form

$$(x,v_1,v_2,\ldots,v_{k-1})$$

or

$$(v_2,v_3,\ldots,v_k,x)$$

where x takes any value from 1 to d/2. To show that the graph has diameter k_1 suppose we want to go from $(v_1,v_2,\ldots,v_k)$ to $(w_1,w_2,\ldots,w_k)$. At the first step go to $(w_k,v_1,v_2,\ldots,v_{k-1})$, at the second step to $(w_{k-1},w_k,v_1,\ldots,v_{k-2})$, and in general at the ith step to $(w_{k-(i-1)}, w_{k-(i-2)},\ldots,v_{k-i})$. After k steps the desired node will be reached.

Suppose we want to go from $(v_0,v_1,\ldots,v_{k-1})$ to $(v_2,v_3,\ldots,v_{k+1})$. By the above method this would be accomplished in two steps, going through $(v_1,v_2,\ldots,v_k)$. However, suppose this last node is not functioning. A different sequence involving only four additional steps can be taken [43]. Let a,b,c, and e take values from 1 to d/2. The sequence is

STEP	NODE	CONDITIONS
0	$(v_0,v_1,\ldots,v_{k-1})$	
1	$(v_1,v_2,\ldots,v_{k-1},a)$	$a\neq x_k$
2	$(v_2,v_3,\ldots,v_{k-1},a,b)$	$b\neq x_k$
3	$(c,v_2,\ldots,v_{k-1},a)$	$c\neq x_1$
4	$(e,c,v_2,\ldots,v_{k-1})$	$e\neq x_1$
5	$(c,v_2,\ldots,v_{k-1},v_k)$	
6	$(v_2,\ldots,v_k,v_{k+1})$	

This method will work as long as there is only one failure. If there are more failures, additional conditions may have to be imposed.

APPENDIX D

ALGORITHM FOR BEST LINK ADDITION IN A TREE NETWORK

This algorithm, from [14], finds the best place to add a link in a tree network in order to minimize the average distance. All pairs of nodes are considered, but the enumeration is organized so that the time to calculate the improvement is proportional to the graph diameter.

The algorithm is given in the paper as a PASCAL procedure. Three data structures are used. A vertex and its associated information is called a node and is defined by:

```
node = record
         adj: ↑ edge;
         desc: 1...n;
         imp,tri,sum: integer
       end
```

The nodes are organized into an array called "a", in which they are numbered from 1 to n. The adjacencies in the network are given by "edges", where

```
edge = record
       vert: vertex
       next: ↑ edge
       end.
```

The procedure is as follows:

```
procedure DFS(w: vertex;d: 1 ...n);
 var p: ↑edge;
begin p :=a[w].adj;
 while p ≠ nil do
  with a[p↑.vert] do { w' =p↑ .vert}
  begin t[d] :=a[w].desc - desc;
   if d mod 2 = 1 then
     begin { d is odd}
      imp :=a[w].imp - a[w].tri
            + 2*desc*a[w].sum;
      trouble :=0;i :=1;j :=(d+3)div 2;
      while j<=d do

       begin
        trouble :=trouble +t[i]*t[j];
        i :=i+1;j :=j+1
       end;
      tri :a[w].tri - trouble;
      sum :=a[w].sum + t[(d+1)div 2]
     end
    else begin { d is even}
       tri :=a[w].tri + t[d div 2]*desc;
       imp :=a[w].imp - tri + 2*desc*a[w].sum;
       sum :=a[w].sum
     end;
   if imp > max then
     begin max :=imp;
       best_vertex.v1 :=v
       best_vertex.v2:=p↑.vert
     end;
    DFS(p↑.vert,d+1);
    p :=p↑.next
   end
 end;
```

By calling DFS(v,1), the best vertex to connect v with becomes the value of the global variable best_vertex.v2. This can be done for all vertices v, and thus the best pair is found. Methods are given in the paper for reducing the number of computations even further.

APPENDIX E

Theorem: There are no Moore geometries when the nodal degree is 2 and the bus degree is greater than 2.

Proof: If such a Moore geometry did exist, it would have

$$1 + 2[(k-1) + (k-1)^2 + \ldots + (k-1)^d] \tag{1}$$

nodes. Now in a Moore geometry the number of nodes times the nodal degree must equal the number of buses times the bus degree. Since the number of buses is integral, we must have twice expression (1) divisible by k. That is, we need

$$2 + 4[(k-1) + (k-1)^2 + \ldots + (k-1)^d] \equiv 0 \pmod{k}. \tag{2}$$

Most of the terms on the left, when the powers are expanded, are powers of k and can be eliminated without affecting congruence (2). Thus we must have

$$2 + r[(-1) + (-1)^2 + \ldots + (-1)^d] \equiv 0 \pmod{k} \tag{3}$$

If d is even, (3) becomes

$$2 \equiv 0 \pmod{k}$$

while if d is odd, (3) becomes

$$-2 \equiv 0 \pmod{k}.$$

In either case, k cannot be larger than 2. ■


FILME
8-8